This is my script for the ViewGraphs. If you run across a set of square brackets, you can choose one of the two phrases based on the group you are talking to. For example a presentation to an academic audience might use courseware *and to a fortune 500 company might use* productivity solutions*.*

The Software Crisis

I would like to talk to you about a crisis. A Crisis that has been going on in the software industry for the last several years. A crisis so severe that it has been called an epidemic. It occurred because software developers have been continually adding features to old programs until they become impossible to manage efficiently. Every time they want to add a new feature they introduce two new bugs, and fixing those bugs creates new bugs.

Falling Productivity

So what we see is a dramatic drop in productivity as you add new features to existing systems. If you look closer, we will find that the software was originally created in an environment appropriate for small projects but inappropriate for large systems. The old tools simply do not cut in when solving problems of a new magnitude.

The NeXT Solution

At NeXT, we feel we have found a solution. A seamlessly integrated development environment based on the principles of object oriented programming. And because we started from scratch we don't have the constraints of some of the other mainstream computer vendors who must support programs and software environments based on software technology developed in the 70s. Our goal is straight forward: make people an order of magnitude more productive creating whatever Applications they need to solve their specific problems. And this does not apply only to programmers. Even people who don't know how to do anything more then point and click should have access to the same tools that until now only programmers could use.

NextStep

We call our solution NextStep. It consists of the middle four layers shown here. It gains additional power by leveraging off of the advanced multi-tasking Mach operating system developed at Carnegie Mellon with complete UNIX ® Berkeley 4.3 compatibility. It has an extremely flexible graphics capabilities because it relies on Display Postscript based window server. Most of our time today will involve a discussion of the next two levels. First we will be discussing the structure of the Application Kit, which is a toolbox of objects we will use to create a complete Application. The we will be demonstrating the Interface Builder. It is used much as one would use a graphics editor to "draw" the layout of the human interface of an application and then connect the objects together. On top of that sits the Workspace. This is the part of NextStep that allows users to do all the manipulation of files without ever having to open a manual. It makes UNIX accessible to even mere mortals. So your applications will be easy to use and consistent with most other applications running in the NextStep environment.

Capabilities Seminar

This class is an overview of NextStep. Since we don't have time to cover all the details, we will stick to trying to explain the "Big Picture" concepts. We would like you to leave here today with an understanding of the **capabilities** of the system, and what **potential** it has assisting you at rapidly creating your own *[courseware | applications]* . So you can see my goal here is to excite and inspire you to create you own custom solutions for your specific tasks.

Seminar Format

Because of the broad background of people who are interested in the NeXT computer, we have partitioned the class into three segments. The first is targeted at a very general audience. We want to appeal to people that have only had a very small exposure to creating their own applications. The second part is for novice programmers, people that might have had some exposure to BASIC or other beginning languages. The third section is targeted at more experienced programmers. Although we have set aside time for short questions during the first sections, I will also be around after the last session for people with detailed questions such as programming in LISP or using the Digital Signal Processor.

The Big Picture

Our task for the first hour will be to build up a foundation for the examples we will be using later on. We will start by defining the scope of our topic and discuss its general advantages. We then go onto a discussion of the more general techniques.

What is an Object?

So what is an "Object" and how do you build a software development environment around it? I am going to double click on the word Object and then use the "Define in Webster" option of the Request menu. The first definition we see is:

something material that may be perceived by the senses

For example a dictionary is an object. If we had a real dictionary here we could see it and touch it. On the NeXT we have a dictionary "Object" which serves the same purpose of a paper dictionary. But it is, in a sense an **abstraction** of a traditional dictionary. Object oriented programming the process of creating abstractions of these things. OOP is a **process** of creating computer based analogs of real word objects. It is a process that is different from the traditional process of software construction, because you always start your design with the highest level of abstraction. You then partition the problem into lower levels of abstractions. You will get to a point where their are already existing objects for you to use to model your problem or the problem will be clear enough that you can create your own objects. Once you partition your problem you carefully control how the objects communicate. This helps you preserve this decomposition.

Plant Example

For example, say we had the problem of modeling plants. The object oriented programmer might first partition the plant into structures such as leaves, stems and flowers. These objects would then also be analyzed further until the problems could be represented by lower level data structures.

Low Level Data Structures

In contrast, many other design methods start by using these lower level structures and keep assembling them until they start simulating the higher level structures. They have a collection of data structures and a collection of algorithms and they build programs by combining them it whatever way best matches their problem.

Benefits of OOP

What are some of the advantages of Object Oriented Programming? Of of the most important is that if you do partition your problem correctly you can re-use other objects. These can be objects that you have developed previously, your colleges have developed, they can be public domain objects available via the Internet, they can be objects provided by a third party software developer, or they can be objects that are provided as part of the operating system on the computer system you use.

Another advantage is that you minimize the amount of work that has to be re-done if your partitioning does not work out. This is because people often spend a great deal of time doing low level detailed work that must be thrown out after they get to a high level and realize they have to re-partition their model.

One of the hidden benefits of the clear partitioning of the problem into objects is that the structure of the code tends to be much more self documenting. If all the code to manipulate an object is together, you know **why** that code is there and what its function is. So if you are looking at other peoples objects, you can expect to spend a lot less time looking through documentation.

Another big win is a very crisp division of user interface objects form the core routine objects. As we will see later, this is very evident with NextStep, where all the user interface routines are available on a pallet.

Many people assume that the objects in a single program must all be created with the same language. But as we will see later all objects communicate in a language independent manner. This means that you can easily mix objects written in C, LISP, FORTRAN and DSP assembly language all together in the same binary file. So if you have a problem that requires a great deal of numerical analysis, you can use a language that can easily be vectorized like FORTRAN. If you have a problem that is symbolic in nature, LISP might be the correct choice for you. If you have a problems that requires data acquisition and analysis, we include a large library of objects that are written in DSP assembly language. The point is that you should always solve a problem using the language that most efficiently solves the problem, and OOP allows you to easily do that.

This last general point is that careful partitioning of your problem allows you to do distributed computing. For example in the BreakApp program, every time the ball hits the wall you hear sound like a plucked string vibrating. This is actually a being done by a the DSP. You might also notice that Mathematica has a preferences menu that allow you to run the kernel on another processor. So you can run your programs on the main 68030, the DSP, on processors that are later added using the current three empty NextBus slots, on a compute server down the hall, or on a file sever a thousand miles away.

Four OOP Techniques

We are now going to start covering the four basic techniques used in Object Oriented Programming that make these benefits possible.

Encapsulation

Encapsulation has various names. Computer scientists often refer to it as information hiding or data abstraction. But in general all these terms mean grouping your 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 access the data in the object. One rule is that if you develop and object, you **only** let people see or change your internal data using the procedures you provide with that object. What that means is that the creator has all control of how others access this object and therefore the creater is responsible for their completeness and correctness.

When you create a mental image of the objects, you might find it helpful to create a mental image of a box with a thick brick wall around it. Inside the box we have the data structures such as integers, floating point numbers, strings, and other more complicated structures such as linked lists. We call these the **instance variables**. 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.

Benefits of Encapsulation

Once you start encapsulating your data you will find that you can quickly control the data types of 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 where it is much easier to isolate. This will dramatically cut down your time spent with the debugger and greatly enhance you programms final reliability.

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

Once you have an object that performs some specific function, you can then create a symbolic abstraction of that object using a "view" of it on the screen. The connections to this can then be done with NextSteps connection based programming tools.

Views of Subroutine Libraries

By creating these views, a user now can manipulate the views of your object and integrate it with other objects. When they make a connection to one of your objects, NextStep will ask them which of the access methods they would like to use. What this means is 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 createors 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 it the only people who could drive a car were the people who could assemble an internal combustion engine. We certianly wouldn’t have the traffic problems we have today. But what we have for cars is a simplified user interface, a steering wheel, a break and a gas peddle. OOP gives us this same easy to interface to traditional subroutine libraries. So we will see that the number of people that are creating applications whith these tools will grow exponentially for the next several years.

So before 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 and click.

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 Inheritence Tree

The structure is very similar to an evolutionary 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 diffacult for beginners using Interface Builder because both classes and instances are represented by small windows that are very close together.

When say that Humans are a **super-class** of Students and that Humans are a **sub-class** of mammals.

Inheritence (continued)

When we create a new class, we inherit all the instance variables as well as methods defined in its super-class.

To be continued.... - Dan