NXApp, Winter 1993 (Volume 1, Issue 1).

An Informal Approach to Object-Oriented Design

written by **Scott Ritchie**

Changing from structured programming to object-oriented programming requires a change in the way you view design. A number of simple tools and methodologies are available to help you start to design and build object-oriented applications.

Times have changed. If you don't understand how to design and develop object-oriented software, you just don't understand how to program anymore. So figure on investing a lot of time and money retraining yourself and your staff in the latest object-oriented design methodology. Not!

Object-oriented programming is extremely powerful, but the task of writing software hasn't changed as much as many people believe. We're still trying to apply tools to the problems that we face in our endeavors, whether modelling financial futures, improving cellular communications, or writing an insanely great interactive game. What has changed, though, is that we now have a new set of tools. Does this mean we programmers are all in for some major re-programming ourselves? I claim it doesn't. If you understand structured programming you will, with experience, have no trouble creating object-oriented software.

To change to an object-oriented methodology, you need to look at design issues slightly differently. You need to take a step back, unlearn a little, and resume talking about software in more human terms again. This approach makes so much sense that if you're programming for the first time, you may wonder how people accomplished anything with older technology.

This article presents an informal design methodology, one that has been successfully applied by thousands of object-oriented programmers. It uses an example program designed with NEXTSTEP in mind, but the concepts presented here can apply to other object-oriented development environments.

Of course, there are many excellent books and seminars available that give the topic of design and analysis a great deal more attention, and this approach is not intended to replace them. However, if you were to give yourself a week to learn object-oriented programming, the hour-and-a-half that you'd spend here should be enough to help you build successful applications from scratch.

WHY OBJECT-ORIENTED DESIGN ANYWAY?

When someone asks you what kind of work you do, do you ever say ªI'm a translatorº? As programmers our primary job is to translate from the ªproblem spaceº of the real world problem into the solution spaceº of the computer platform.

If someone asked you to translate an ancient Chinese manuscript into English, you might not have the skills to do it, but you'd probably guess that it could be done with only some loss of the
manuscript's original subtlety and beauty. On the other hand, if you were asked to express a poem in predicate calculus, you'd probably bet there was no way to do it successfully. Simply stated,
if there's too much distance between the domains of the problem space and the solution space, the translation is destined to fail.

Programming computers is a task of translation. If we reduce the ªdistanceº in the translation,
we reduce the likelihood of introducing errors. This is why we aren't all still

programming in assembler: Compiler technology allowed high-level structured languages to reduce this distance. Object-oriented languages close the gap even further.

High fidelity software

Think about the changes that compact disc technology has brought to recorded music over earlier media. The recording industry is focused on faithful reproductions of the original performances. Success here is largely due to advances in integrated circuit (IC) technology: powerful re-usable hardware components with clearly defined interfaces for their inclusion in larger systems.

Object-oriented design attempts to model a real world problem in terms the people working on it use and understand. The designer looks for objects in the problem space and models them at an appropriate level of abstraction. Brad Cox, the principle author of the Objective C language, refers to these objects as *software-ICs*. Analogously, these software components make it much easier to produce high fidelity softwareÐsoftware that is faithful to the original problem description.

An expert from the problem area, known as the *domain expert*, is an important member of an object-oriented design. The domain expert provides the details of the situation being modelled. The programming team tries to gain expertise and understanding in the domain from the expert. This reverses an old pattern in software. Instead of limiting the design by educating users about the capabilities of the platform, we gain expertise from a domain expert and apply it to the solution.

By letting us use abstraction and encapsulation, objects allow us to talk about the solution space in real world termsÐterms that are provided by users, technical writers, instructors, and other domain experts. Because users, designers, and implementers are all speaking the same language, the solution is robust. Resulting programs actually solve users' problems and are much easier to document, learn, and understand. Thus object-oriented design allows us to save on the greatest hidden costs in software development.

Design methodology

Design is a disciplined approach we use to invent a solution for some problem. It provides a path from requirements to implementation.

Traditional engineering approaches have a design phase followed by an implementation phase. Often these tasks are carried out by separate groups with less than ideal communication. There seems to be a notion that at the end of the design phase, the designers will drop off the Frozen Specification with the implementers and board planes for their vacations.

What's wrong with this picture? Designers could gain tremendously from the experience of the implementers. Without ongoing feedback from the implementers, the design runs the risk of being difficult, if not impossible, to realize. Often new insights are gained as the code is being written that could simplify or otherwise improve the design.

Φ1_2ΠησεΕνγινεερινγ.επσ ←

Figure 1: *Traditional engineering approaches have a design phase followed by an implementation phase.*

I assert that all good software is written at least twice. Business is dynamic. In some sense, a program is successful if users want to continue to change and expand it. Competitive advantage lies, then, in quick response to changing business needs.

Object-oriented design emphasizes the incremental and iterative development of systems. The process is neither top-down nor bottom-up; rather, it's ªround-trip.º The object designer starts with a simple framework of interacting objects and then continually refines and fleshes out the model. Each model within a design describes a certain aspect of the system.

As much as possible, we build new models upon old models that we already have confidence in. With software, unlike with automobiles, you want objects with high odometer readings; their heavy use assures you that their design is re-usable and reliable.

Φ2_ΙτερατιϖεΔεσιγν.επσ ←

Figure 2: *Object-oriented design emphasizes incremental, iterative development.*

Objects

Even if you have never programmed a computer, you already understand how object-oriented software works. It's based on the way that people naturally handle complexity: by abstracting and grouping. If you have programmed in a high-level structured language like C, then you have a further advantage in learning a language that implement these principles.

So what do objects bring to the party? By encapsulating both data and the procedures that operate on that data, objects provide us with the ability to abstract. Only certain details about the object are important to usersÐthe rest are implementation details and are appropriately hidden in a protected way. As designers, we are free to determine what level of abstraction is appropriate for the task. Indeed, this is often one of the most important design decisions to be made. I believe you'll come to see this flexibility as an advantage, not an ambiguity.

Objects with similar traits or behavior are grouped together in a class. Classes also inherit their traits and behavior from an ancestral class called a *superclass*. This allows us to describe new classes in terms of objects that are already understood, written, and tested.

OBJECT-ORIENTED DESIGN PROCESS

Typically, the process of object-oriented design loops through the four steps shown in Figure 3. This is an iterative process. Identifying classes and objects usually causes us to refine and improve on the semantics of existing objects. Implementing classes often leads to discovering new classes and objects that simplify the design.

Before you start: What seems to be the problem?

As designers and implementers of software, we often miss one of the most likely causes of failure: Failure to clearly identify the problem to be solved. When presented with vague or complex specifications and desires, we have an obligation to push back and seek clarity, simplicity. This may very well be the hardest step in any project, but it's the most importantÐif you haven't clearly stated the problem, your system isn't going to meet the demands of its specifiers.

For example, suppose you were to start a project with the following problem statement: ªBuild an application that presents a very simple card file of customer information.º

The application maintains a list of customers.

The characteristics of the current customer are displayed in text fields in a window.

Customer characteristics include at least a name and a unique identifier.

To add a new customer, you choose New from a menu.

To delete the current customer, you choose Delete from a menu.

To search through the list, you click Next and Previous buttons.

The projects you work on are probably much more complex than this. However, although this example is simple, the solution will use the same basic architecture as even the most complicated graphical applications: Model-View-Controller. Each object in the application can be thought of as falling into the Model, View, or Controller category. All layers are represented in this problem.

When you begin to approach your problem, simplify. Attempt to strip away everything that isn't absolutely essential to the basic purpose of the application as you begin your first iterations toward a solution. In doing so, you will discover the fundamental framework within a week or so, rather than months later. If your problem specifiers can't see the value of this simplification process, point out that if you don't succeed in getting this framework developed first, the advanced features they want are ultimately doomed to failure anyway.

For instance, in this example, is the choice of a list as the data structure optimal? Will restricting searches to linear operations satisfy users? The answers are almost certainly ªNoº on both counts, but creating this simplified description does allow you to identify the objects at a high level of abstraction. To complete the application, you'll follow this design process and increase the level of detail on each iteration, eventually addressing the more complex questions.

Figure 3: *Phases of the iterative object-oriented design process*

What objects are needed?
The first phase of this design process is identifying classes and objects. You begin by listing potential objects for your solution to the problem. Also keep track of all the actions that need to be performed, even if the ªactorsº themselves are not yet determined. At a later time you can assign each of these responsibilities to a specific class of object.

Identifying classes and objects is largely a process of decomposition: dividing the problem into smaller and smaller parts, each of which you refine independently. It involves both discovery and invention. Through discovery, you'll recognize the objects in the problem domain as well as their interactions. Through invention, you'll devise new objects and interactions to regulate how existing objects behave.

Objects are often one of the following:

Tangible things: Cars, telemetry data, pressure sensors

Roles: Mother, teacher, politician

Events: Landing, interrupting, requests

Interactions: Loan, meeting, intersection

Here's a surprisingly simple way to derive candidate objects: Underline the nouns and verbs in an informal description of the problem. The nouns represent candidate objects. The verbs represent operations on them.

You can find a number of candidate objects in our sample problem. Objects in the view layer include TextFields, Window, Menu, and Buttons. Objects to represent the data or model layer include List and Customer. The Controller layer binds the view to the model for this specific problem. The problem statement indicates a need for an Application object.

With further iterations in the design process you can decompose these objects into component parts. Knowing which objects are already available from

NEXTSTEP and other kits can influence the design. All of the objects in the view layer are provided by any self-respecting graphical toolkit. The Application Kit further provides a reusable Application class that you can just instantiate and use. Realizing this, you may wish to invent a separate Controller class to provide specific control for this design.

Without worrying about which objects provide them, you can also find some responsibilities in the problem statement: adding and deleting customers, entering and editing text, setting and getting customer name, displaying the current customer, and searching for next and previous customers.

What does each object look like?

The second phase of the design process is identifying class and object semanticsÐdeciding what each represents. This is largely a process of abstraction, in which you emphasize some of the system's details and suppress others.

To best accomplish this, look at each class from the point of view of an outside observer; view it from the perspective of its interface. Identify the things you can do to each object and then the things that each object can do to other objects. One useful technique is to write a script for each object, defining its life cycle from creation to destruction and including its characteristic behaviors.

For example, you can use *Class Responsibility Collaborator cards* (CRC cards) to explore what each object can do and how it affects other objects. Write the information on basic 3 x 5 index cards. The cards' flexible, informal character encourages experimentation, and their small size discourages over-burdening any single class. Each card represents a different object in the design; each holds a class name, a list of responsibilities the class will manage, and a list of collaborating objects this object uses to do its work. List any private data or methods on the back of the card, to further emphasize their protected nature.

Φ4_ΧΡΧχαρδσ.επσ ←

Figure 4: *CRC cards for the Customer and Controller classes*

How are objects related?

The third step in the iterative process is identifying class and object relationships. This is the primary difference between a library and a kit: The objects are important, but full understanding involves the relationships. This is new territory for structured language programmers!

To identify the relationships among classes, you want to look for patterns. The two most common
patterns arise between objects that have either of the following characteristics:

They have similar attributes and/or behavior.

They contain other objects.

Classes that share structure and behavior have an *inheritance* relationship. Objects that are *composed* contain other objects and need instance variables to hold them.

Look at the proposed classes for generality in behavior and state, or similarities in component parts. Inheritance captures patterns in the objects' classes. This helps generalize the components being created and increases their reusability and reliability. You implement general classes once and create specializations of them (*subclasses*) to fulfill the program's needs.

In the card file problem, there must be a Customer object. Similar programs could manage a classroom roster of students or display a genealogy. These programs would require Student and FamilyMember objects, which would be very similar to Customer. Figure 5 shows how you might create a more general Person class capturing the similarities between these three more specialized classes. Although the general class isn't strictly needed for this application, the Person class provides generally usable functionality that you might be able to use elsewhere. Creating objects for a kit of your own builds ªobject equityº that can significantly reduce future development effort.

Φ5_ΟβφεχτΧλασσεσ.επσ ←

Figure 5: *The general class Person captures the similarities of more specific subclasses.*

When you use composition instead of inheritance, you create an object that contains other objects. In this way you can create multi-function objects by combining specialized classes. Their complexity is made manageable by their composite nature. Composed classes often tend to be creators; they usually instantiate the objects they contain when they themselves are initialized.

To inherit or to compose

Suppose you decide to create a new model object called CardFile that keeps track of the current customer. Figure 6 shows how you could create CardFile, either by composition or inheritance. If CardFile is composed, it contains a pointer to an instance of the List class called customerList. Alternatively, CardFile could inherit from the List class and specialize it.

Creating a class by composition rather than inheritance implies that the class is more general and just happens to contain objects of a certain type. Composition allows more flexibility in the type of objects contained. For example, CardFile might initially use a list as the data structure. Later, you could change it to use a more efficient structure like a hash table or btree.

Φ6_Χονστρυχτ+Ινηεριτ.επσ ←

Figure 6: *The CardFile class could use either composition or inheritance.*

Using inheritance indicates creating an object that is a more specialized version of its superclass. If CardFile inherits from List, it must always remain a type of list. On the other hand, inheritance generally requires less work. Since CardFile is itself a List, it no longer needs the instance variable customerList. The method **addObject:** is inherited from the List class and hence CardFile doesn't even need a new **addCustomer:** method.

Implementation

The fourth step in the process involves taking an inside view of each class and writing the code. This is seldom the last step. Unless the problem is trivial, you'll return to the first step and repeat the cycle at a lower level of abstraction.

From the CRC cards and the problem description, you make a first pass at writing the code. The responsibilities of a class often appear as methods. The collaborators represent *outlets*, instance variables that refer to other objects. Any private data can be kept in additional instance variables.

An Objective C header file for the Customer class might look like this:

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

@interface Customer:Object
{
  char *name;
  int uid;
}
- setName: (char *)newName;
- (char *) name;
- setUid: (int)newUid;
- (int) uid;

@end
```

Because of an object-oriented language's strict separation of interface and implementation, you can now reliably divide up large projects among a group of programmers. Each programmer can have complete freedom in deciding how to best implement each piece of the design.

At this point, although the implementation is very general, it's faithful to the problem description. It can be tested and used. This rapid prototyping is another powerful benefit of object-oriented programming and design.

Completing the application is now a matter of increasing the level of detail. You can turn to the domain expert at this point to fill in these greater levels of information, restarting the design cycle.

BUILDING ªOBJECT EQUITYº

Don't be surprised if designing with objects initially takes longer than you expected; it's a sign that you're probably doing the right thing. Careful design is always time-consuming, even with structured languages. Other languages just don't require you to do it, nor do they aid you the way an object-oriented language does.

Simplify whenever possible. People tend to make things more complicated than they need to be. Begin with the design specifications. Take full advantage of the tools that objects provide, especially abstraction. Consider the reusability of objects and refine them when possible so you can add them to your own kit.

In this age when there are enough object-oriented seminar flyers to paper your den, I encourage you to try this informal, natural approach to design. Have fun with it. Get users, writers, and other domain experts involved early. You're building the ªobject equityº you need to enjoy the factor-of-ten increase in productivity often touted by sales people and object-oriented pundits. n

Scott Ritchie is a Developer Trainer at NeXT. You can reach him by e-mail at **Scott_Ritchie@next.com**. Special thanks to Randy Nelson, Steven Asbury, and Scott Weiner, who contributed to this article.

OBJECT-ORIENTED DESIGN REFERENCES

Booch, Grady. *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1991. ISBN 0-8053-0091-0. *A classic text on design. Not a light read, it nevertheless is one of the best design books by virtue of its author's extensive experience in the domain.*

Coad, Peter, and Edward Yourdan. *Object-Oriented Analysis*, 2nd ed. Englewood Cliffs, NJ: Yourdan Press, 1991. ISBN 0-13-629981-4.

Coad, Peter, and Edward Yourdan. *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdan Press, 1991. ISBN 0-13-630070-7.

Cox, Brad J., and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*, 2nd ed. Reading, MA: Addison-Wesley, 1991. ISBN 0-201-54834-8. *A ªwhyº rather than ªhowº book about objects. Brad Cox was the major author of the original Objective C language.*

Goldberg, Adele, and David Robson. *Smalltalk-80, The Language and Its Implementation*. Reading, MA: Addison-Wesley, 1983. ISBN 0-201-11371-6. *The original text on what is generally regarded as the original object-oriented language. Model-View-Controller architecture and the origins of much of Objective C are here.*

Humphrey, Watts S. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989. ISBN 0-201-18095-2.

Jacobson, Ivar, et al. *Object-Oriented Software Engineering*. Reading, MA: Addison-Wesley, 1992. ISBN 0-201-54435-0.

Martin, James, and James J. Odell. *Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992. ISBN 0-13-630245-9.

Mullin, Mark. *Rapid Prototyping for Object-Oriented Systems*. Reading, MA: Addison-Wesley, 1990. ISBN 0-201-55024-5. *A good book for consultants or anyone interested in designing software that focuses on the user interface and involves the customer or problem specifier. Not NeXT-specific but valuable for its real-world insights and usable techniques.*

NeXT Computer, Inc. *NEXTSTEP Development Tools and Techniques*. Palo Alto, CA: Addison-Wesley, 1992. ISBN 0-201-63249-7. *See especially the tutorials in Chapters 15 through 18. Also available on-line with NEXTSTEP Developer*.

Pinson, Lewis J., and Richard S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Reading, MA: Addison-Wesley, 1991. ISBN 0-201-50828-1. *Has a very good object-oriented design section. Deals with Objective C but only Release 1 of NEXTSTEP.*

Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991. ISBN 0-13-629841-9.

Wirfs-Brock, Rebecca, et al. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990. ISBN 0-13-629825-7.

Ð*Scott Ritchie and Stephen Asbury*

MODEL-VIEW-CONTROLLER

Model-View-Controller is a standard architecture for interactive object-oriented applications. It was first presented in the Smalltalk-80 language, and it works very well for NEXTSTEP applications.

A Model-View-Controller design has three layers. Any object in an application falls into one of the layers.

ModelViewControl.eps ¬
An example of objects in the Model-View-Controller layers

The objects in the Model layer tend to be the crown jewels of a company. They represent the data and the algorithms to manipulate it. Examples include classes that represent data structures and connections to databases or networks. Ideally these objects are designed to be portable to a variety of platforms; they are highly likely to be distributed.

The objects in the View layer represent the user interface. Examples include windows, menus, buttons, and text fields. These objects are tied to a particular interface like NEXTSTEP but are highly reusable to provide consistency between applications. There are usually a lot of these objects in any given application. They often come straight from the application kit.

The Controller layer objects provide the binding between the Model and View layers. They contain application-specific code and are hence the least likely to be reusable. They stand between the other layers, and as a result Model objects don't need to know about user interfaces and View objects needn't be aware of data structures. The Controller objects usually represent the bulk of the custom code in an application, but they account for a small percentage of the objects.-*SR*

Next Article NeXTanswer #1503 **Branching Out With Dynamic Loading**
Previous article NeXTanswer #1506 **Spreading the Wealth: DO and PDO**
Table of contents
http://www.next.com/HotNews/Journal/NXapp/Winter1994/ContentsWinter1994.html