{ Chapter 3 }

The Programmer's Approach

This chapter might appear to be the most ambiguous and vague in the book. However, it's probably the most valuable chapter because it can save you time. Effective programming requires more than just brute force, math skills, and technical ability (which are all important); you must also be efficient. For any programming task, you can do it the hard way or you can do it the easy way. This chapter will help you find the easy way. I don't want to suggest that there are always shortcuts that make anything easy. Only that avoiding overly complex solutions can save you a lot of grief.

This chapter explains conventional programming philosophies to help you

- Write specifications
- Prototype your creations quickly for initial review
- Develop good style
- Keep code and data separate

Specification

It might sound cavalier, but after you finish a detailed specification, 95% of your work is through. When a client asks me whether something is possible, I always answer that if he can describe it in detail, I can program it. That's all a specification is—a detailed description of exactly how the Flash movie is to appear and perform. A good specification can take a lot of time and work, but when it's finished it serves as the blueprint from which to work. It's like creating an outline for a term paper. After you know where you're headed, it's just a matter of filling in the outline.

One person's idea of the necessary level of detail might vary from another's. But the more detail, the better. When you invest additional work upfront, it not only saves time down the road, but it reduces the chances of rework because everyone involved presumably reads the "spec" and raises necessary objections early. Another value of a specification is that it makes estimating total cost easier because the task is clearer.

As you'll see in the following section on prototyping, the problem in writing a super-detailed specification is that you will fail to fully describe the final program because a written spec's form is different from the final media. That is, it's impossible to describe the colors in a painting using just words or the sound of a song without some kind of musical device—there's a translation error. This doesn't mean you should simply forgo the specification process entirely. Rather, just write enough detail to get rolling. In addition, be sure to leverage previous work of your own, and the work of others. For example, part of your specification might say that you're going to build something "like the project we did last year," but with these differences. Do whatever you need to make it clear. Include tables, figures, and pictures—whatever helps.

Writing specifications takes practice. An interesting exercise to improve your specification writing skills is to go back and look at any description or specification you were given early in a project that is now finished. Identify the types of details that were missing or would have helped were they given to you. Without providing a lesson on how to write specifications, let me simply say those details make a big difference.

Prototyping

A good specification makes you more efficient because it defines the course you'll take before you start. Even though this means that you want to wait to start programming, there is one type of programming that can start early, even while you write the specification: prototyping. A *prototype* is a quick and dirty sample that's created in Flash simply to get an idea of how the final project might look and feel. No matter how great a specification on paper might be, it will never compare to the real thing. It's sort of like learning to fly an airplane. You can read all the books, use a flight simulator, even hang out at the airport—but eventually, you'll have to actually go up in the airplane and fly! Prototypes are the best "simulation" of your final movie because they are produced in Flash.

One way to produce a great site involves first roughly defining the objectives and then starting the prototyping right away. You make a few quick prototypes and then analyze the results. Let everyone play with the prototypes so that they can get a feel for the direction the site is taking. Then go back to the drawing board and elaborate on your specification. This cyclical process (defining, prototyping, redefining, and so on) might seem slightly inefficient, but it usually results in a better end product.

If ever there were a place where being sloppy is alright, it is while prototyping. The goal while prototyping is to quickly get something up and running. The prototype doesn't have to be pretty. Just check out Figure 3.1, which shows an early prototype of part of a larger project.

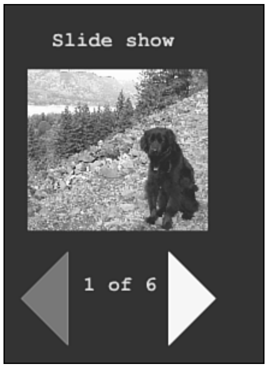


Figure 3.1 *A prototype doesn't have to be pretty—it just has to communicate the idea. You can even use a picture of your dog as a placeholder.*

Your prototypes will not only look rough, but you can allow the programming to slip a bit, too. Let's look at a few prototyping techniques that—during any other stage of programming—could be considered bad style.

Hard Wiring

Hard wiring is generally a big no-no. *Hard wiring* is using an explicit value instead of a variable or reference to other values. For example, imagine that your program is supposed to display a message "Welcome <User>," where "<User>" would be replaced by whatever name the user used when logging in. Instead of actually doing the work to display the user's name dynamically, you could hard wire the screen to read "Welcome Phillip" and then any time you demonstrate the prototype, you make sure to log in as "Phillip." Of course, if you logged in differently, the program would still display "Welcome Phillip" because it's hard wired. This is perfectly acceptable because likely everyone can just imagine how this will work in the final project. Sometimes you'll want your prototype to really do the work—especially if you're sending the prototype to a client who might not understand that it's just a prototype. Depending on who is judging the prototype, you can do more or less hard wiring as appropriate. Just remember the term "hard wiring" because in later chapters I'll refer to it as a bad thing—ideally everything is dynamic.

Pseudo-Code

Unlike hard wiring, pseudo-coding is always a good thing. The only problem is that you'll need to replace all your pseudo-code eventually. Pseudo-coding is the process of writing scripts using your own words, not the ActionScript language, as instructions. Then you can replace your completed pseudo-code with components of ActionScript. The truth is that really detailed and clear pseudo-code can be quickly and easily translated to functioning ActionScript. I often say that if you can pseudo-code well, you can get a monkey to clean it up ("cross all the *T*s and dot all the *I*s," as it were). You might find that you can clearly state what you want your interactive movie to do, but you need help from an experienced programmer to translate your pseudo-code. But the process of pseudo-coding actually makes you a better programmer by forcing you to sort out the details of the task you're solving.

Pseudo-code should be very detailed and written in clear English. That is, you want to say *everything* necessary, but you don't need to use any words from the ActionScript language. For example, imagine you plan to program a button that will convert a dollar amount from one field and display the equivalent value in Euros in another field. (By the way, the preceding description is suitable as part of a specification—it provides enough information to start programming.) The pseudo-code for such a button might look like this:

```
When user presses button
    dollars=text in field
    exchangeRate=.5
    euros=dollars multiplied by exchangeRate
    euros=euros rounded off to two decimal places
    put euros into another field
end
```

After you know a little bit of ActionScript, you can easily translate this pseudocode into a working script. The first step, however, is to sort things out in your own words.

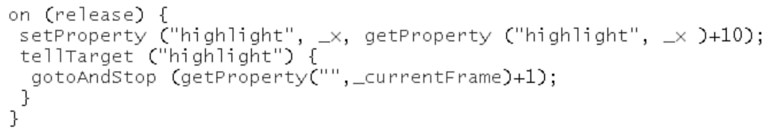
Good Style

If this chapter is ambiguous, this section is downright subjective. *Good style* means programming in a way that's easy to maintain. Your code should be easy enough for anyone to understand. Not because others need to see what you've programmed (which could happen), but so that you can quickly interpret what you've produced when you need to make adjustments or fix bugs. It's easy to get carried away trying to build something and ignore good housekeeping practices. Before you know it, your code resembles a plate of spaghetti (hence the term *spaghetti code*). In fact, haste makes waste, so you should always try to follow the rules of good style.

Even though the value of good style is easy to understand, the concept itself is subjective. Here are a few characteristics of good style; call them *rules* if you want.

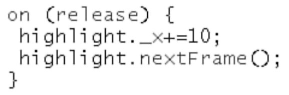
Less Is More

Consider that every line of code you read has to be translated in your mind. You have to figure out what it really means. The fewer lines you must read the better. Generally, any time you can do something in fewer steps or less code—do it. It's almost never too late to use less code. For example, I often start programming and then come up with a better (more concise) solution while implementing the original idea. Even if it means going back and starting over, it's usually worth the resulting compact code. Compare the two code segments in Figures 3.2 and 3.3—both achieve the same effect, but the code in Figure 3.3 is much more concise.



```
on (release) {
 setProperty ("highlight", _x, getProperty ("highlight", _x )+10);
 tellTarget ("highlight") {
  gotoAndStop (getProperty("",_currentFrame)+1);
 }
}
```

Figure 3.2



```
on (release) {
 highlight._x+=10;
 highlight.nextFrame();
}
```

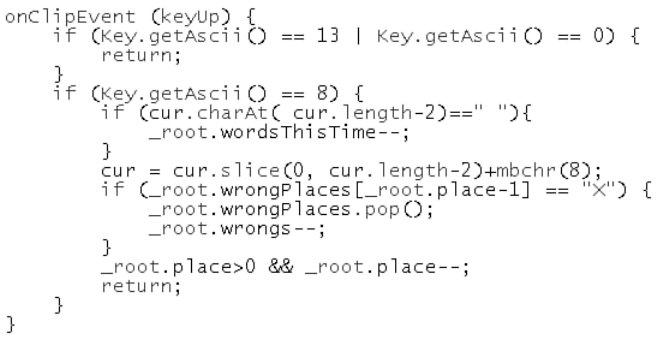
Figure 3.3 *The script in Figure 3.2 achieves the same effect as the script in Figure 3.3, but with unnecessary complexity.*

You can take this rule too far. The appeal of concise code should not outweigh legibility. It's easy to get carried away and end up with code you can't even read. I would never fault a finished piece of code that worked—so, really, that's the number one priority. Second, your code must be maintainable by you. Remember to write code that you can read. For example, in Chapter 4, "Basic Programming in Flash," you'll learn that `count++;` is equivalent to `count=count+1;` Although the latter takes more typing, it might be easier for you to read. By all means, use what you understand. If this occasionally means that your code is a little bit wordier, so be it. Take it one step at a time and you should see your code shrinking in size.

Comments

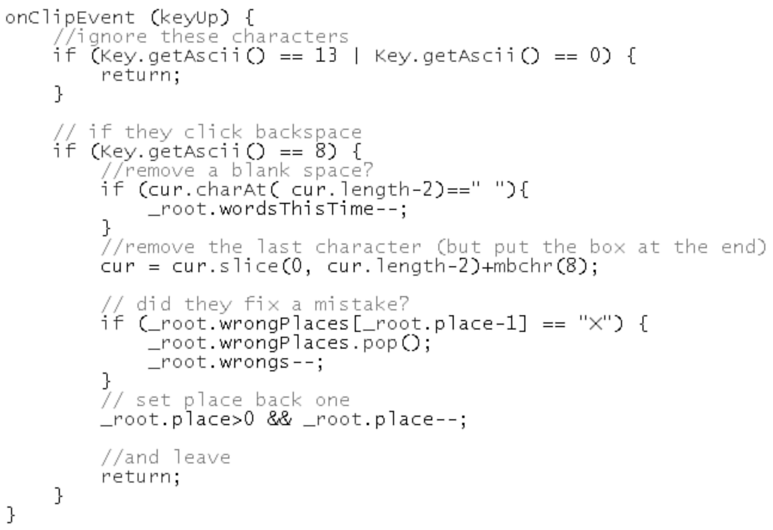
Comments are lines of code that are ignored by Flash. Text preceded by `//` is ignored. Actually, if you start a block of text with `/*`, all lines are ignored until `*/` is reached. The idea is that you can write notes to yourself (or anyone reading your code) that explain—in normal English—what's going on in the code. Actually, you'll often find that comments help you discover bugs. You might see a comment that says `//loop through all the answers` and then notice that the code doesn't really do that!

I suppose that I'm a bad boy because I often don't fully comment my code until right after I get a program running. However, it's important for me not to delay this step because I will forget everything about the code days after writing it. Without comments, code is much more difficult to interpret. So, just take the time to comment your code, even if it's after you've finished and when the incentive to do so is reduced. Compare the uncommented code in Figure 3.4 to the same code with comments in Figure 3.5. Even though you might not understand the details of the code, if there were a problem, you could easily identify the portion containing the problem.



```
onClipEvent (keyUp) {
    if (Key.getAscii() == 13 | Key.getAscii() == 0) {
        return;
    }
    if (Key.getAscii() == 8) {
        if (cur.charAt( cur.length-2)==" "){
            _root.wordsThisTime--;
        }
        cur = cur.slice(0, cur.length-2)+mbchr(8);
        if (_root.wrongPlaces[_root.place-1] == "X") {
            _root.wrongPlaces.pop();
            _root.wrongs--;
        }
        _root.place>0 && _root.place--;
        return;
    }
}
```

Figure 3.4



```
onClipEvent (keyUp) {
    //ignore these characters
    if (Key.getAscii() == 13 | Key.getAscii() == 0) {
        return;
    }

    // if they click backspace
    if (Key.getAscii() == 8) {
        //remove a blank space?
        if (cur.charAt( cur.length-2)==" "){
            _root.wordsThisTime--;
        }
        //remove the last character (but put the box at the end)
        cur = cur.slice(0, cur.length-2)+mbchr(8);

        // did they fix a mistake?
        if (_root.wrongPlaces[_root.place-1] == "X") {
            _root.wrongPlaces.pop();
            _root.wrongs--;
        }
        // set place back one
        _root.place>0 && _root.place--;

        //and leave
        return;
    }
}
```

Figure 3.5 *Before being commented, the code in Figure 3.4 is difficult to understand. Figure 3.5 shows how a few comments can make things clearer, even if you don't understand the underlying code.*

Finally, comments are of great assistance while creating a prototype. Instead of building *everything,* you can simply place a comment that says something like `//check their answers here` and then come back later to actually write the code that does. This technique also exposes errors in logic flow. Remember that specifying exactly what a Flash movie is supposed to do is most of the work. A comment can be a way of specifying the tasks that need to be implemented.

Magic Numbers, Constants, and Variables

A "magic number" is an explicit value used within a formula. For example, to calculate the page count for any chapter in this book, I use this formula: characters/1900=pages. I know there are approximately 1900 characters per printed page. Of course, if the margins or page size were different I'd have to use a different magic number than 1900 in my formula. An example of a constant is pi. To calculate the area of a circle, use pi times radius squared (πr^2). Generally, magic numbers should be avoided because they're dangerous. At a minimum, they should be commented.

Consider what happens if I use my magic number for characters per page in many places and then the book layout changes—maybe we change the paper size. I would need to replace every instance of 1900 with the new number. The ultimate solution in this case is to use a variable (discussed in Chapter 4) like a constant. At the very beginning of my movie, I could establish a variable "charsPerPage" as 1900 (`charsPerPage=1900;`). Then, instead of using `1900` in several locations, I could use `charsPerPage` instead of my constant. If `charsPerPage` were to change, every instance would reflect the change. Compare magic numbers to a `gotoAndPlay(2)` Action (where 2 is the magic number). A better solution is to use a frame label (which you can think of as a constant), as in `gotoAndPlay("loopFrame")`. If you move the "loopFrame" label, you won't need to go and fix your scripts in any way.

It's very easy to think a magic number will *never* need to change, so it doesn't seem worth the effort to create a variable that can be used like a constant. In reality, magic numbers are not evil. It just takes a bit of foresight to realize whether such a number could potentially change—in which case, you should use a variable instead.

Repeated Code

To put it simply: Every programming task should appear only once in your movie. If you have the same code in two places, that means you'll have twice the work to make updates or fix bugs. You'll learn ways to achieve this—such as keeping scripts in the library, in functions, or external to the movie itself—but for now, just make sure that anytime you copy and paste code, a bell rings in your head to notify you there must be a better way.

You'll probably develop more techniques that exemplify good style. Remember, it's subjective and based on personal preference. Although there are definitely methods that *should* be employed by all programmers, you only need to acquire skills as you become comfortable. I know that if I looked at anything I programmed even just a few months ago, I'd question the approach I took—but that's because I'm always improving. If you waited until your skills were perfect, you'd be waiting a long time. Just jump in, but take the time to be self-critical so that you can improve.

Code Data Separation

All programmers should strive to keep code (that is, the programming scripts) separate from the data (or the project-specific content such as text and graphics). By keeping code separate from data, you enable all your programming efforts to transfer easily to other projects. Similarly, when you want to make a major change to the content—say, redo the entire project in a different language—you just need to replace data without touching (or breaking) the code. It's a great concept that is sometimes difficult to achieve.

Imagine a factory that produces furniture with a wide selection of fabric upholstery. Likely the upholstery (think "data") is kept separate from the furniture and padding (think "code") until an order is placed. The benefit of code data separation (in this analogy) is that the factory can easily produce furniture as its customers request it and never have additional stock that's already upholstered. Applying this to Flash isn't much different. Assume that your Flash site has graphic buttons that display a floating tool tip whenever the user places his cursor over the button. If you kept the code (the script that makes the tool tip appear) separate from the data (the actual text or words that appear in the tool tip), you could easily translate this to another language by replacing the text for the tool tips. Ideally, you would keep *all* the text for all the tool tips in one location to make translation that much easier. The main idea is that you want to be able to make significant changes to either the code or data without affecting the other.

You can think of code data separation as a form of modularization. There are other forms of modularization—including Flash's `LoadMovie()`, which enables you to play separate .swf files within a larger movie. Modularization has many benefits in addition to those mentioned for code data separation. For one thing, by modularizing your Flash movie, users won't have to wait for the entire site to download. They can selectively download just the portions in which they are interested. Also, modularizing makes working with others easy and efficient. Consider that if you just had one master file for the entire site, only one person could work at a time. So, there are a ton of benefits to code data separation and other kinds of modularization. Without providing a lot of details now, just realize that throughout this book I'll try to emphasize solutions that exhibit such modular attributes.

Summary

This chapter explored the attributes that make up the "programmer way." In my experience, it seems that programmers tend to fit the same profile. For example, they often work in darkened offices that lack windows and subsist on soft drinks.

You don't have to become a geek to be a good programmer. Just concentrate on the approach discussed in this chapter. Try to develop a good style by striving to write concise code with lots of comments and avoid magic numbers and repeated code. Realize, too, that your programming style should continually improve. The best programmers in the world know they have room to improve further.

The process you undertake can also make programming easier (and better). Creating a specification and quickly producing prototypes might seem like additional up-front work, but they will save you time later. Finally, always try to separate code from data. In no time, you'll start "feeling it" and before you know it, you can call yourself a "programmer" with pride.