Chapter 14. Organizing Straight-Line Code

Contents

This chapter turns from a data-centered view of programming to a statement-centered view.

It introduces the simplest kind of control flow: putting statements and blocks of statements in sequential order.

Although organizing straight-line code is a relatively simple task, some organizational subtleties influence code quality, correctness, readability, and maintainability.

1. Statements That Must Be in a Specific Order

The easiest sequential statements to order are those in which the order counts.

Here's an example:

Example 14-1. Java Example of Statements in Which Order Counts

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

- Unless something mysterious is happening with this code fragment, the statement must be executed in the order shown.
- The data must be read before the results can be calculated, and the results must be calculated before they can be printed.

2. Statements Whose Order Doesn't Matter

You might encounter cases in which it seems as if the order of a few statements or a few blocks of code doesn't matter at all. One statement doesn't depend on, or logically follow, another statement.

But ordering affects readability, performance, and maintainability, and in the absence of execution-order dependencies, you can use secondary criteria to determine the order of statements or blocks of code.

The guiding principle is the Principle of Proximity: Keep related actions together.

Making Code Read from Top to Bottom

As a general principle, make the program read from top to bottom rather than jumping around.

- Experts agree that top-to-bottom order contributes most to readability.
- Simply making the control flow from top to bottom at run time isn't enough.
- If someone who is reading your code has to search the whole program to find needed information, you should reorganize the code.

Here's an example:

Example 14-8. C++ Example of Bad Code That Jumps Around

```
MarketingData marketingData;
SalesData salesData;
TravelData travelData;

travelData.ComputeQuarterly();
salesData.ComputeQuarterly();
marketingData.ComputeQuarterly();

salesData.ComputeAnnual();
marketingData.ComputeAnnual();
travelData.ComputeAnnual();

salesData.Print();
travelData.Print();
marketingData.Print();
```

- You have to look at and think about every line of code in this fragment to figure out how `marketingData` is calculated.

Example 14-9. C++ Example of Good, Sequential Code That Reads from Top to Bottom

```
MarketingData marketingData;
```

```
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();

SalesData salesData;
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();

TravelData travelData;
travelData.ComputeQuarterly();
travelData.ComputeAnnual();
travelData.Print();
```

This code is better in several ways:

- References to each object are kept close together; they're "localized."
- The number of lines of code in which the objects are "live" is small.
- And perhaps most important, the code now looks as if it could be broken into separate routines for marketing, sales, and travel data. The first code fragment gave no hint that such a decomposition was possible.

[Cross-Reference
A more technical definition of "live" variables is given in "Measuring the Live Time of a Variable" in Scope.]

Grouping Related Statements

Put related statements together:
They can be related because they operate on the same data, perform similar tasks, or depend on each other's being performed in order.

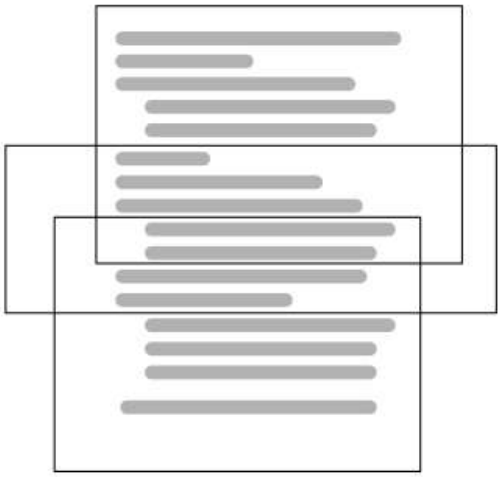
An easy way to test whether related statements are grouped well is to print out a listing of your routine and then draw boxes around the related statements. If the statements are ordered well, you'll get a picture like that shown in Figure 14-1, in which the boxes don't overlap.

F14xx01

Figure 14-1

If the code is well organized into groups, boxes drawn around related sections don't overlap. They might be nested.

If the code is organized poorly, boxes drawn around related sections overlap: you might want to refactor the strongly related statements into their own routine. Figure 14-2.



F14xx02

Figure 14-2

If the code is organized poorly, boxes drawn around related sections overlap.

Checklist: Organizing Straight-Line Code

- Does the code make dependencies among statements obvious?
- Do the names of routines make dependencies obvious?
- Do parameters to routines make dependencies obvious?
- Do comments describe any dependencies that would otherwise be unclear?
- Have housekeeping variables been used to check for sequential dependencies in critical sections of code?
- Does the code read from top to bottom?
- Are related statements grouped together?
- Have relatively independent groups of statements been moved into their own routines?