Why You Should Use Routines... Routinely

As an undergraduate computer science student, I thought that the main reason to create new routines, instead of leaving all the code in one big routine, was to avoid duplicate code. This is undoubtedly the most popular reason for creating a routine, and it’s a good one. Similar code in two routines is a warning sign. David Parnas says that if you use copy and paste while you’re coding, you’re probably committing a design error. Instead of copying code, move it into its own routine. Future modifications will be easier because you will need to modify the code in only one location. The code will be more reliable because you will have only one place in which to be sure that the code is correct.

That is one good reason to create a routine, but it hardly makes a complete list.

**Why Else?**

There are many additional reasons to create routines, and many of them are more important than avoiding duplicate code.

**Reducing complexity**

The single most important reason to create a routine is to reduce a program’s complexity. Create a routine to hide information so that you won’t always need to think about it. Sure, you need to think about it when you write the routine. But after it’s written, you should be able to forget the details and use the routine without any knowledge of its internal workings. Other reasons to create routines—minimizing code size, improving maintainability, and improving correctness—are good ones, but without the abstractive power of routines, complex programs would be intellectually impossible to manage.

**Limiting effects of changes**

Isolate areas that are likely to change so that the changes’ effects are limited to a single routine or, at most, a few routines. Create your designs so that the areas most likely to change are the easiest to change. Such areas include hardware dependencies, input-output formats, complex data structures, and business rules.

**Hiding sequences**

Hide the order in which events happen to be processed. If the program typically gets data from the user, then gets auxiliary data from a file, neither the routine that gets the user data nor the routine that gets the file data should depend on the other routine being performed first. Design the system so that either could be performed first, then create a routine to hide the information about which happens to be performed first.

**Improving performance**

Having code in one place, inside a single routine, means that a single optimization benefits all the code that calls that routine. It makes it practical to recode the routine with a more efficient algorithm or a faster, more difficult language, like assembler.

**Hiding data structures**

Hide data structure implementation details so that most of the program doesn’t need to worry about messy manipulation details, but can deal with the data in terms of how it’s used in the problem domain. Routines that hide implementation details provide a valuable level of abstraction that reduce a program’s complexity. They centralize data structure operations in one place and reduce the chance

Continued on page 94
of errors in working with that data structure. They
make it possible to change the data structure with-
out changing most of the program.

When hiding a data structure, refer to it inde-
pendently of the media it’s stored on. If you have
an insurance rates table that’s so big it’s always
stored on disk, you might be tempted to refer to it
as a “rate file.” When you refer to it as a file, however,
you’re exposing more information about the data
than you need to. If you ever change the program
so that the table is in memory instead of on disk,
the code that refers to it as a file will be incorrect,
misleading, and confusing. Try instead to make ac-
cess routine names independent of how the data is
stored, and refer to the abstract data type: “insur-
ance rates table,” for instance.

**Hiding global data**

If you need to use global data, at least hide its
implementation details. Working with global data
through access routines provides several benefits.
You can change the structure of the data without
changing your program, and you can monitor ac-
cesses to the data. The discipline of using access
routines also encourages you to think clearly
about whether the data is really global; it might
be more accurate to treat it as local to a class, mod-
ule, or routine.

**Promoting code reuse**

Code put into modular routines can be reused in
other programs more easily than the same code
spread across a large routine.

**Planning for a family of programs**

If you expect a program to be modified, isolate
the parts you expect to change within their own
routines. You can then modify the routines without
affecting the rest of the program, or you can put the
changes into completely new routines instead. For
example, several years ago I managed a team that
created a family of programs used by our clients to
sell insurance. We had to tailor each program to
each specific client’s insurance rates, report format,
underwriting rules, and so on. But many parts of the
programs were similar: data input routines that took
information about our clients’ customers, routines
that stored and retrieved information in a customer
database, routines that computed total rates for a
group, and so on. The team modularized the pro-
gram so that each part that varied from client to
client was put into its own module. The initial pro-
gramming for the whole system took several
months, but after that was done, whenever we got
a new client we merely wrote a handful of new mod-
ules and dropped them into the rest of the code. A
few days’ work, and voila! Custom software!

**Improving readability**

Putting a section of code into a well-named rou-
tine is one of the best ways to document its func-
tion. Instead of reading a series of statements like

```c
if ( Node <> NULL ) then
  while ( Node.Next <> NULL ) do
    Node = Node.Next
  wend
  LeafName = Node.Name
else
  LeafName = ''
endif
```

you can read a statement like

```c
LeafName = GetLeafName( Node )
```

The new routine is so short that all it needs for doc-
umentation is a good name. Using a function call
instead of eight lines of code makes the routine that
originally contained the code less complex and au-
tomatically documents it.

You can use the same technique to document
complicated Boolean tests. Move the code that per-
foms the test into its own function, out of the main
flow of the code, then choose an understandable
function name. Both the main flow of the code and
the Boolean test will be clearer.

**Improving portability**

Isolate use of nonportable operations to explicit-
ly identify and isolate future portability work. Nonportable operations include nonstandard lan-
guage features, hardware dependencies, operating
system dependencies, and so on.

**Isolating use of nonstandard language functions**

Most languages contain handy, nonstandard ex-
tensions. Using them is a double-edged sword: they’re useful, but they might not be available
when you move to different hardware, a new ven-
dor’s implementation of the same language, or
even a new version of the language from your cur-
rent vendor. If you use nonstandard extensions,
build routines of your own that act as gateways to those extensions. Then you can replace the vendor’s nonstandard routines with custom-written routines of your own if needed.

Isolating complex operations

Complex operations include complicated algorithms, communications protocols, tricky Boolean tests, and operations on complex data—all of which are prone to errors. When you detect an error, having the code to which it applies contained within a single routine makes the error easier to diagnose and fix than if parts of the complex operation were spread throughout the program.

**The Greatest Invention in Computer Science**

Aside from the invention of the computer, the routine is arguably the single greatest invention in computer science. It makes programs easier to read and understand. It makes them smaller; imagine how much larger your code would be if you had to repeat the code for every call to a routine. And it makes them faster; imagine how much harder it would be to make performance improvements in similar code used in a dozen places. In large part, the effective use of routines makes modern programming possible.

Continued from page 93