You’ve no doubt heard the saying, “A man with a watch knows what time it is. A man with two watches is never sure.” This is often called Segal’s law, which credits a certain Lee Segall (his name actually has two l’s) with making this observation sometime in 1961. Others have found a still earlier use in a 1930 newspaper article.1

If you’re an engineer, you probably wouldn’t be confused by having two watches. If both seemed to be working, you’d take the average of the two readings. If one stopped, you’d use the other, and if both stopped, you’d look for a third. But this would work only if you had both watches in your possession. It would be different if you had one watch and had to coordinate with someone else who had the second (unsynchronized) watch. In that case, you’d want both parties to use a single common reference for the time.

The necessity of a single common reference is enshrined in a software-engineering principle that’s so clear you need to hear it only once. Once you’ve heard it, you’ll forever suffer the frustration that you’re not quite able to live up to it.

The SPOT, or Single Point of Truth, principle says that you should specify key pieces of information in one and only one place in your code. Any related information should be derived from that single source and not stored separately from it. The key information could be a data item, a procedure, or an interface definition. The benefit is that if you need to change something, you can change it at that one place and be sure that all derived information is updated as well.

The principle is popular enough that it has several names. For instance, you can find references to Single Source of Truth (SSOT), Single Version of Truth (SVOT), Don’t Repeat Yourself (DRY), and the more ferocious Duplication Is Evil (DIE).

Spotting Bugs
The dangers of not adhering to the SPOT principle are easy to show. I’ll start with a simple example that can bite, especially in larger code bases.
It can cause bizarre run-time errors that can be hard to pin down.

Consider the following little C program, which I call spot1.c:

```c
#include <stdio.h>

float a;
extern void fct(void);

int main(void)
{
    a = 3.14;
    fct();
    printf("a=\%f\n", a);
    return 0;
}
```

The program uses one externally defined function, `fct()`, that will access the globally declared variable `a`. After the external call to `fct()` completes, the program simply prints the value of `a`

The function `fct()` is defined in the file `spot2.c`, which declares `a` as an external variable and accesses it as follows:

```c
extern int a;

void fct(void)
{
    a = 314/100;
}
```

Now you’ve noticed, mostly because there are just a few lines of code to look at here, that the second file got the type of `a` slightly wrong. Surely the compiler will catch this, right? Let’s find out:

```
$ gcc --Wall --pedantic -o spot1 spot1.c spot2.c
$ ./spot
a=0.000000
```

For good measure, we compiled this program with all warnings enabled, and in pedantic mode so that we can also learn about the smallest-possible problems the compiler might detect. But the compiler issues no warnings. The problem is that the inconsistency in the type of `a` can be detected by only the linker, not the compiler itself, so it slips through, with potentially grave consequences. I’ve also tried several commercial static-source-code-analysis tools on this small example program, and none reported the coding error either.

As another example of how developers all too often ignore the SPOT principle, consider this code fragment:

```c
const float pi = 3.14159265358979;
#define ten_pi 31.4159265358979
```

The first line defines the constant variable `pi` and initializes it, up to some suitable level of precision. Then, possibly in a different file, or a few thousand lines away from this first declaration, we might see a macro definition for a value that’s 10 times the value of `pi`. Clearly, we now have two sources of information for the value of the single numerical constant π. Don’t be surprised if the Boolean expression

```c
(ten_pi/10 == pi)
```

now yields the result `false`. A better solution would be to replace the macro definition with, for instance,

```c
#define ten_pi (10*pi)
```

which restores the single point of truth.

We programmers routinely violate single points of truth.

As another example of how developers all too often ignore the SPOT principle, consider this code fragment:
different name than in the original declaration, you’ve indirectly broken the SPOT principle. It’s not the computer that gets confused in this case; it’s your peer who must maintain and possibly repair your code when problems show up, long after you’ve moved on.

Truth in Documents
Despite the SPOT principle’s appealing simplicity, it’s violated almost everywhere. Out of curiosity, I tried to check how much duplication there was among the files on my computer. That was easy enough. Using the md5sum tool, I computed 128-bit MD5 hashes of all regular files in my file system. I then sorted the output by hash value to detect duplicates.

Shockingly, I discovered that over the years, I had accumulated close to a million files. What were they all for? I was also shocked to find that approximately 85,000 of those files had at least one duplicate. Now, somewhat reassuringly, approximately 10,000 of those files matched only because they were empty or contained just a single line-feed character (it happens). Also, many of the other matching files were autogenerated by-products of various tools. More worrisome was that a few hundred of the duplicated files were larger than a megabyte. That small exercise was an easy way to reclaim disk space.

Truly duplicate information isn’t the real problem, though. Duplication merely sets a trap for a more vicious problem. The trap closes when one or more of the copies are updated, each with slightly different new information. Consider those three copies of your phone and address list that live on your work computer, your laptop, and the flash drive in your pocket. Most likely, these three files differ, each having been updated at separate points.

Once you lose the single point of truth, restoring it can be exceedingly difficult.

How do you merge them back together? Detecting and resolving this type of near duplication is much harder. Once you lose the single point of truth, restoring it can be exceedingly difficult.

The near-duplication problem occurs also in industry. In the design of large software systems, system engineers often document the original design requirements in spreadsheets. Those spreadsheets later find their way into a requirements database, but the original spreadsheets are rarely discarded at that point as now obsolete duplicates. From the spreadsheet or database, yet more copies are created in software design documents, PowerPoint presentations, and test requirement documents. After enough time passes, nobody knows which version of a document is current or can trace back what changes were made in all the versions. Sometimes this means that if you want to know how a requirement is satisfied, it’s best to look in the code itself. This then means that, ultimately, the software developer and tester, not the system engineer, decide how to interpret a requirement. As you can imagine, this doesn’t always end well.

Cloning
Large organizations often reuse software modules from one project to another, frequently by cloning and modifying the earlier code. Defects in the original version don’t always reach the clones, and vice versa, because the new projects are typically under different chains of management and work from a different set of design documents. And yes, all those new design documents, more often than not, also start their lives as clones of earlier project documents that are modified along the way.

What happens in all these cases is that an informal social process subtly interferes with a more rigorous engineering process. Many organizations have learned the hard way that reuse can’t be an afterthought. For software to become eligible for reuse, it should be designed and documented for reuse from the start. You’d never rewrite a library of trig functions or sorting routines without good reason. If the library is properly designed and mathematically rigorous, it shouldn’t need to be rewritten. If, nonetheless, a defect is discovered in a key library, any updates should be directed to just a single master copy of the code. A library that’s designed for reuse in this way can indeed form a reliable single point of truth across multiple projects.

Interface Design
I once heard someone say that the Unix system architecture is a good
example of an object-oriented design. What that person probably meant was how Unix standardizes the interfaces to peripheral devices and defines the default interface between application programs. In both cases, the interface is that of a simple byte stream. Unix programs are often designed to read text input from the standard input channel and write text to the standard output channel. Because of this uniform interface definition, at least in the early days most Unix applications could read the output of any other application and produce output that other applications could process. The definition of a standard interface lets you build complex applications from small parts that can be connected without much effort.

Calling this approach object-oriented might be a misnomer, though. You could just as easily argue that it’s the opposite. By standardizing interfaces on the least common denominator in the system—text—the OS avoids having to define specialized interfaces for each different type of object that may be handled. The elegance of this standardization shows most clearly in the handling of peripheral devices, which are by default instrumented to produce or consume byte streams. This shields users from the details of the device interfaces and lets them think about each device as if it’s just a regular file. The details are “hidden” behind the interfaces. But the key idea isn’t data hiding either, as Dave Parnas described it. The key idea is the localization of data: creating single points of truth.

References

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.