Points of Truth

Gerard J. Holzmann

Jet Propulsion Laboratory, California Institute of Technology

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 referred to as "Segal's law," which credits a certain Lee Segall (his actual name has two l's) with making this observation sometime in 1961. Others have found a still earlier use in a newspaper article from 1930 [1]. Segal's law is related to a principle of software engineering called the "Single Point of Truth."

If you're an engineer, you would probably not be confused by having two watches. If both watches seem to be working, you would take the average of the two readings. If one of the two has stopped, you use the other, and if both watches have stopped you better look for a third. But this only works if you have both watches in your possession. It is different if you have one watch and you need to coordinate with someone else who has a second (unsynchronized) watch. In cases like this you really want both parties to use a single common reference for the time.

The necessity to have a single common reference is enshrined in a principle of software engineering that is so clear that you only need to hear it once. Once you've heard it, you will forever suffer the frustration that you're not quite able to live up to it. This is the SPOT principle, which is short for the "Single Point of Truth."

The SPOT principle says that key pieces of information should be specified in one and only one place in your code. If other information is based on it, it should be derived from that single source and not stored separately from it. The key piece of information could be a data item, but it can also be a procedure or an interface definition. The benefit is that if you need to change something, you can change it in one and only one place in the code, and you can be sure that all derived information will get updated as well.

The principle is popular enough that it is known under several different names. For instance, you can find references to "Single Source of Truth," "Single Version of Truth," or "Don't Repeat Yourself" (with the acronym DRY), and more ferociously also to "Duplication Is Evil" (with a less-pleasing acronym).

The dangers of not adhering to the SPOT principle are easy to show. I'll start with a very simple example that can bite especially in larger code bases. It can cause bizarre types of runtime errors that can be hard to pin down.

Consider the following little C program, let's call it **spot1.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, called **fct**, that will access the globally declared variable named **a**. After the external call completes it simply prints the value of **a**.

The function **fct** is defined in a different file called **spot2.c** that declares the variable **a** as an extern, and accesses it as follows.

```
     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 variable a slightly wrong. Surely the compiler will catch this, right? Let's find out.

```
     $ gcc –Wall –pedantic –o spot 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 learn also about the smallest possible problems that the compiler might detect. But, the compiler issues no warnings. The problem here is that the inconsistency in the type of variable **a** can only be detected by the linker, not by 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 of those report the coding error either.

The real root of the problem is that we have specified the type of variable **a** twice, violating the single point of truth principle. One solution is to introduce a header file, say **spot.h**, and move the external declaration of variable **a** into that header file:

```
     #ifndef SPOT_H
     #define SPOT_H
          extern float a;
     #endif
```

This header file is now included at the start of both **spot1.c** and **spot2.c**, which allows the compiler to now catch the bug reliably. Note that the variable is now declared in only one place in the program. The file **spot1.c** also contains the definition of the variable, which allocated memory for it, but it does so in a place where the type of **a** in the definition can be checked against the type provided in the earlier external declaration. The external declaration acts as the single point of truth for the type of variable **a**.

As another example of a way that the SPOT principle is ignored all too often, consider this code fragment:

```
const float pi = 3.14159265358979;

#define ten_pi   31.4159265358979
```

The first line defines a constant variable named 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 may see a macro definition for a value that is ten 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

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

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

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

which preserves the single point of truth.

There are many ways in which we programmers routinely violate single points of truth in our code. If, for instance, you take the address of a variable or a function, and in doing so make the object that is pointed to accessible under a different name than used in the original declaration, you've indirectly broken the single point of truth principle. It is not the computer that gets confused in this case, it is your peer who has to maintain and possibly repair your code when problems show up, long after you've forgotten all about it.

Truth in Documents

Despite its appealing simplicity, the single point of truth rule is violated almost everywhere. Out of curiosity, I tried to check how much duplication there is among the files on my computer. This is easy enough to do. I used the **md5sum** tool to compute 128-bit MD5 hashes of all regular files in my file system. I then sorted the output by hash value to detect duplicates. When I did this, the first shock was to discover that over the years I have accumulated close to a million files. What are they all for? The second shock was that my little duplication test found that about 85 thousand of these files had at least one duplicate elsewhere in the file system. Now, somewhat reassuringly about ten thousand of those files matched only because they were empty or contained just a single linefeed character (it happens), and many other of the matching files were auto-generated byproducts 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 some disk space on my system.

Truly duplicate information is not the real problem though. Duplication merely sets a trap for a related problem that is worse. 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 on the flash-drive in your pocket. Most likely all three of these files differ, each having been updated at separate points. How do you merge them back together? This type of near-duplication is much harder to detect and resolve. Once you lose the single point of truth it can become exceedingly difficult to restore it.

The near-duplication problem occurs also in industry. In the design of large software systems the original design requirements are often documented by system engineers 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 the 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 different versions. Sometimes this means that if you want to know how a requirement is satisfied, it is best to look in the code itself. This then means that it is ultimately the software developer and the tester who decide how a requirement is interpreted, and not the system engineer. As you can imagine, this does not always end well.

Cloning

In large organizations not only documents but also software modules are often reused from one project to another, often by cloning and modifying the earlier code. Defects found in the original version do not always reach the clones, or vice versa, because the new projects are typically under different chains of management and they 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 interferes in subtle ways with a more rigorous engineering process. Many organizations have learned the hard way that reuse cannot be an after-thought. For software to become eligible for reuse it should be designed, and documented, for reuse from the start. Note that, without good reason, you would never rewrite a library of trigonometric functions, or a library of sorting routines. If the library is properly designed, and mathematically rigorous, it shouldn't ever 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 and nowhere else. A library that is designed for reuse in this way can form a reliable single point of truth across multiple projects.

Interface Design

I once heard someone say that the original design of the Unix operating system is an example of object-oriented design. Meant was probably the way in which Unix standardizes the interfaces to peripheral devices, and the way in which it defines the default interfaces 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 to write text to the standard output channel. Because of this uniform interface definition, most Unix applications can read the output of any other application, and produce output that other applications can process. The definition of a simple standard interface allows one to build complex applications from small parts that can be connected without much thought.

Calling this an object-oriented approach may be a misnomer though. It could just as easily be argued that it is the opposite. By standardizing interfaces on the least common denominator in the system, i.e., text, the operating system avoids the need to specialize interfaces based on the types of objects being handled. The elegance of this standardization shows most clearly in the handling of peripheral devices, which are by default instrumented to produce or consume simple byte streams. This shields the user from the details of specific device interfaces, and allows them to think about each device as if it was just a regular file. To details are “hidden” behind the interfaces. But the key idea is not really data “hiding,” as described so well in the early papers of Dave Parnas [2]. The key is the “localization” of data: the creation of a single point of truth.

References

[1] http://www.barrypopik.com/index.php/new_york_city/entries/P3400/

[2] Parnas, D.L. (December 1972). "On the Criteria To Be Used in Decomposing Systems into Modules". *Communications of the ACM* **15** (12): pp. 1053–58.

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.