U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

History of Software Engineering

Software Engineering 2018

Bruno Carvalho	Diogo Yaguas	Tiago Castro
up201606517	up201606165	up201606186

October 6, 2018

Contents

The Law of Leaky Abstractions

Synopsys

The Law of Leaky Abstractions[8] is an article written by Joel Spolsky, now CEO of StackOverflow, with the intent of demonstrating to the reader that programming abstractions are not perfect and are merely ways of making a programmer's life easier, they should not be relied on completely when aiming to be a good programmer.

To introduce the concept of **leaky abstractions**, Joel focuses on data exchange protocols using TCP as an example – a way to transmit data that is reliable and is a key piece in the engineering of the Internet which we rely on every single day – which is an abstraction based on IP – an unreliable method that doesn't guarantee the arrival of the messages or their integrity. TCP attempts to provide a complete abstraction of an underlying unreliable network, but sometimes, the network *leaks* through the abstraction and you feel the things that the abstraction can't quite protect you from. To show how this abstraction may fail, he uses the hypothesis of a "pet snake that chewed through the network cable leading to a computer, and no IP packets can get through, then TCP can't do anything about it and the message doesn't arrive". This is the point where the key phrase of the whole article is presented:

> "All non-trivial abstractions, to some degree, are leaky."

Learning how to use the basics of the base on which an abstraction is built is needed when learning and using the said abstraction. An example of this need is the use of strings in C++:

> "When I'm training someone to be a C++ programmer, it would be nice if I never had to teach them about char*'s and pointer arithmetic. It would be nice if I could go straight to STL strings. But one day they'll write the code "foo" + "bar", and truly bizarre things will happen, and then I'll have to stop and teach them all about char*'s anyway."

Summing up the whole point of the text, the law of leaky abstractions means that whenever somebody comes up with a wizzy new code-generation tool that is supposed to make us all ever-so-efficient, you should learn how to do it manually first, then use the wizzy tool to save time so that when the abstraction fails, you know how to fix it. This way, you will be a better, more complete programmer.

> "When you need to hire a programmer to do mostly VB programming, it's not good enough to hire a VB programmer, because they will get completely stuck in tar every time the VB abstraction leaks."

Analysis: Impact on Software Engineering

The analysis presented in the article can be lifted to present a more general outlook on software development, particularly the structure of programming languages. When we look at the vast territory of programming languages out there, we can classify them as low and high-level pretty easily. We do so based mostly on how complex are the abstractions that they provide for elementary and direct interactions with the hardware. These abstractions are built on top of other abstractions, from lower-level languages, forming layers of abstractions – often many of them for the highest-level languages. This is analogous to a code database written in a single language (like an operating system), a layered communication protocol, and more.

The abstractions provided by the lower level layers are inevitably leaky, as they are slaves to the hardware limitations of the machine they are running on (not to mention legacy issues, conventional disputes, etc.). An optimal, perfect high-level language wouldn't have any of these limitations, but the abstraction that is built on top of layer cannot fundamentally be rid of the limitations of the lower one - the abstraction *leaks* - and at best it can transform them into a different restraint, considered more negligible or easier to handle: demanding more or unbounded time/memory to complete some operation, demanding boilerplate and verbose code…

The developer who is oblivious to this and faces one such problem stemming upwards from a low-level layer, will attempt to solve it only at the top level, ignorant of its causes, and is doomed to fail. He discovers he needs to learn the functionality of the lower layers to find how to circumvent and solve his problem, or ultimately accept he needs to take an entirely different approach.

The law of leaky abstractions pushes the learning curves upwards, making them ever so steeper in every area of software engineering.

AT&T 1990 Long Distance Network Crash

Overview

The infamous AT&T long-distance telephone network collapse started at 2:25pm on Monday, January 15th 1990, and lasted for about nine hours until engineers managed to patch and stabilize the system.[2] Approximately half of all calls placed on the network failed to go through, returning a *busy* signal to the caller.

AT&T lost more than $60 million in unconnected calls, not to mention the amount of revenue lost by businesses heavily reliant on the telephone network, such as airline reservation systems, hotels and delivery companies. Furthermore, this came to have a huge negative impact on the company's reputation, as the network had been advertised and was viewed by the public as the *epitome of reliability*.[7]

Long Distance Network Operational Model

Let's go back to 1990 to see how the network functioned *on a good day*.

AT&T's long distance, countrywide telephone network was built on the backbone of a system comprised of 114 computer-operated electronic *switches*, called 4ESS, scattered throughout the US – each capable of directing about 700.000 calls per hour.

Parallel to this network of switches there were 14 *long-distance routes* capable of holding and transmitting the actual phone calls.[2]

Whenever a call was issued, the phone's local network looked for the receiver's location. When out of reach it requested the wide network a long range transmission, and the call was delegated up to one of the *switches*, which then scanned all the long-distance routes to establish the connection. Simultaneously, it sent the receiver's telephone number through a parallel signaling network, which checked for alternate routes involving other minor networks, and, in particular, determined if the receiver's local network could handle the call – meaning it could be delegated down.

If the network found that the destination switch along the chosen long-distance route was busy, and it also couldn't delegate the call to a local network, it returned to the caller a *busy* signal, releasing the line. However, if the destination switch signaled available, a reservation was made and a connection was established. The entire process took no long than $4 - 6$ seconds,[2] and allowed for real-time communications from coast to coast.

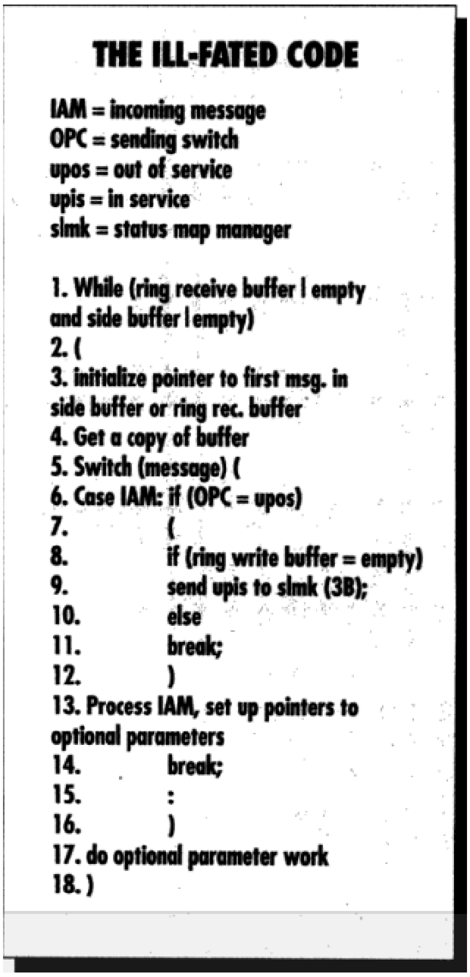
Analysis: January 15th Incident

It all started on the New York switch, which performed a routine self-test and found it was nearing its load limits. Following procedure, the switch, which we will call A, sent a message – a *congestion signal* – over the network indicating it would not take any more requests until further notice, and promptly performed a four second reset.

The problem arose when A came back online. Prior to a recent software update – aimed at speeding up switch intercommunications – A would send a message through the network indicating it was back online and ready to receive and send new traffic – a *revival signal*. After a very short delay, it would start sending out said traffic. However, with the recent software update, there were no revival signals anymore, and A proceeded to send traffic immediately. Another switch, say B, would learn that A was back online only when traffic from A reappeared on the network.[4]

From the perspective of switch B, learning that A was back online required it to update some internal records – presumably the same way as before the software update.

However, recall that A was under stress, and nearing its maximum throughput of about one call every five milliseconds. The situation was better after the reset (as some traffic had been redirected to other switches) but not by much, so it was still sending traffic to the network at unparalleled speed. For switch B, this implied that the first message it received from A, causing it to update its records, was quickly followed by a second message again from A, something which did not happen before the update. This set the precedent for a software bug, which had gone unnoticed in testing, to start wrecking havoc in the system.



THE ILL-FATED CODE

```
IAM = incoming message
OPC = sending switch
upos = out of service
upis = in service
slmk = status map manager

1. While (ring receive buffer I empty
and side buffer I empty)
2. (
3. initialize pointer to first msg. in
side buffer or ring rec. buffer
4. Get a copy of buffer
5. Switch (message) (
6. Case IAM: if (OPC = upos)
7.            (
8.            if (ring write buffer = empty)
9.            send upis to slmk (3B);
10.           else
11.           break;
12.           )
13. Process IAM, set up pointers to
optional parameters
14.           break;
15.           :
16.           )
17. do optional parameter work
18. )
```

The smallest bug breaks the network

The defect was found in a C program which contained a misplaced *break* statement (line 11).[2,7] The piece of pseudo code shown is invoked whenever a switch receives a message.[7] For B, updating its internal records amounts in part to lines 8 and 9 in the code – writing to the *ring write buffer*, which behaves like a one-slot message queue to another subsystem in the same switch.

When B got the second message, **the buffer had not been cleared yet**, so the program would proceed into the else clause and break out of the switch case. But this meant that line 13, to be run unconditionally, **was not executed**. This had severe implications further down the function, namely overwriting the non-empty ring write buffer.

Parallel error correction software would detect this overwrite instantly, and would follow established protocol and shut down switch B, resetting it in a fashion similar to that of switch A. And so A's four second intentional reset became B's four second *unintentional* reset. This cascaded throughout the network, endlessly for nine hours.

Overwriting the ring write buffer is a type of race condition. Consider all possible switches B – some would overwrite the buffer, and thus reset, and some would not. This is what allowed the network to not become completely engulfed in reset madness, and still manage to allow about half of all calls placed on it go through.

Lessons Learned

> "Unsurprisingly, it is not difficult for a simple software error to remain undetected, to later bring down even the most reliable systems."[2]

When we think of high-impact, vast-reaching software bugs, we don't tend to think they could be something as simple as this.

But we know for a fact software is extremely fragile – and, paradoxically, usually quite strict as well, due to the programming languages used. Language strictness prevents common mistakes such as unintended type conversions and errors. Detecting incorrect control flow, however, is the job of an independent tool, specifically a coverage analysis tool. But still the error would surpass the tool's scrutiny – which would show unconditional coverage of line 13, as intended – unless a clever test routine that stressed the switch managed to trigger the race condition.

A more structured programming language would have made this particular defect more likely to pop up at compile time, as the compiler might have detected the race condition, or even impossible to construct. But perhaps the biggest contributor to this catastrophic failure was the routine practice of resetting switches – it would be far better if minor problems could be handled without shutting down an entire switch.

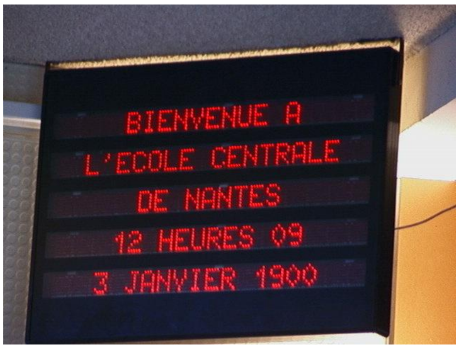
It is this tremendous lack of certainty that plagues software engineering in general. There is no combination of language, tools and testing practices that can guarantee a large software project such as this has no *errors*, not even inconsequential ones. It is up to the developers themselves to follow best practices when they write code and establish guidelines, when they design protocols and platforms, and ultimately when they must confront and repair the errors found in their creations.

Year 2000 Problem

Synopsys

Also known as the millennium bug, it was a problem in the coding of computerized systems that was projected to create havoc in computers and computer networks around the world at the beginning of the year 2000. After more than a year of international alarm, feverish preparations, and programming corrections, few significant failures occurred in the transition from December 31, 1999, to January 1, 2000.

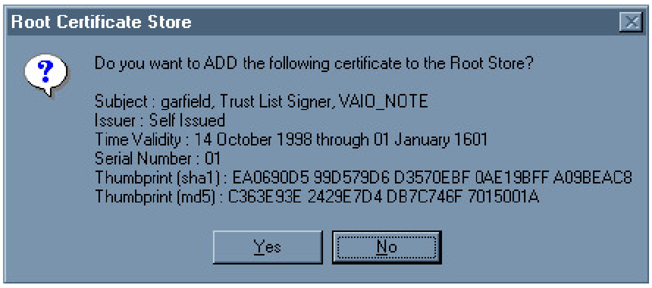
Computer programmers have been in the habit of using two-digit placeholders for the year portion of the date in their software. For example, the expiration date for a typical insurance policy or credit card was stored in a computer file in MM/DD/YY format.



The 2-digit year format created a problem for most programs when "00" was entered for the year. The software does not know whether to interpret "00" as "1900" or "2000". Most programs, therefore, default to 1900. That is, the code that most programmers wrote either prepends "19" to the front of the two-digit date, or it makes no assumption about the century and therefore, by default, it is "19". This wouldn't be a problem except that programs perform lots of calculations on dates. For example, to calculate how old you are a program will take today's date and subtract your birthdate from it. That subtraction works fine on two-digit year dates until today's date and your birthdate are in different centuries. Then the calculation no longer works. For example, if the program thinks that today's date is 1/1/00 and your birthday is 1/1/65, then it may calculate that you are -65 years old rather than 35 years old. As a result, date calculations give erroneous output and software crashes or produces the wrong results.

The Y2K problem was not limited to computers running conventional software, however. Many devices containing computer chips, ranging from elevators to temperature-control systems in commercial buildings to medical equipment, were believed to be at risk, which necessitated the checking of these "embedded systems" for sensitivity to calendar dates.

An estimated 300 billion dollars was spent (almost half in the United States) to upgrade computers and application programs to be Y2K-compliant. As the first day of January 2000 dawned and it became apparent that computerized systems were intact, reports of relief filled the news media. These were followed by accusations that the likely incidence of failure had been greatly exaggerated from the beginning. Those who had worked in Y2K-compliance efforts insisted that the threat had been real. They maintained that the continued viability of computerized systems was proof that the collective effort had succeeded. In following years, some analysts pointed out that programming upgrades that had been part



Do you want to ADD the following certificate to the Root Store?

Subject : garfield, Trust List Signer, VAIO_NOTE
Issuer : Self Issued
Time Validity : 14 October 1998 through 01 January 1601
Serial Number : 01
Thumbprint (sha1) : EA0690D5 99D579D6 D3570EBF 0AE19BFF A09BEAC8
Thumbprint (md5) : C363E93E 2429E7D4 DB7C746F 7015001A

of the Y2K-compliance campaign had improved computer systems and that the benefits of these improvements would continue to be seen for some time to come.

Several very different approaches were used to solve the Year 2000 problem in legacy systems. Three of them follow:

Date expansion

Two-digit years were expanded to include the century (becoming four-digit years) in programs, files, and databases. This was considered the "purest" solution, resulting in unambiguous dates that are permanent and easy to maintain. However, this method was costly, requiring massive testing and conversion efforts, and usually affecting entire systems.

Date re-partitioning

In legacy databases whose size could not be economically changed, six-digit year/month/day codes were converted to three-digit years (with 1999 represented as 099 and 2001 represented as 101, etc.) and three-digit days (ordinal date in year). Only input and output instructions for the date fields had to be modified, but most other date operations and whole record operations required no change. This delays the eventual roll-over problem to the end of the year 2899.

Windowing

Two-digit years were retained, and programs determined the century value only when needed for particular functions, such as date comparisons and calculations. (The century "window" refers to the 100-year period to which a date belongs). This technique, which required installing small patches of code into programs, was more straightforward to test and implement than date expansion, thus much less costly. While not a permanent solution, windowing fixes were usually designed to work for several decades. This was thought acceptable, as older legacy systems tend to eventually get replaced by newer technology.

Software Solutions

In 1996, Rudy Rupak created the Millennium Bug Kit. This freeware solution was one of the first downloadable solutions on the internet at the time and was found in one in four computers and marketed through Planet City Software as Millennium Bug Compliance Kit.

1983 Soviet nuclear false alarm incident

Overview

The year is 1983, a time when the relations between the US and the Soviet Union were severely strained.

Four years before, the Soviet had deployed fourteen SS-20 / RSD-10 theatre nuclear missiles (1979) and consequent NATO response consisting in the deployment of 108 Pershing II missiles ready to target Western Europe.

Since 1981, the United States had been running psychological operations against the Soviet Union to test their radars' capabilities and to demonstrate the US' power. Clandestine naval operations near the Soviet territory and bomber flights to their border several times a week that turned away at the last moment are some of the examples of this type of aggression.

The incident

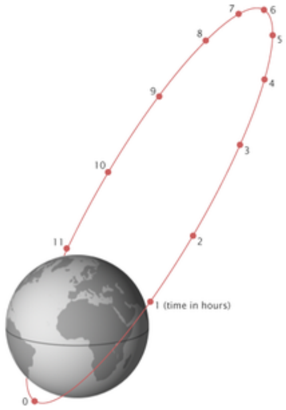
That's why Stanislav Petrov, a lieutenant colonel in Soviet Air Defense Forces and the officer of the Serpukhov-15 bunker in charge of the Soviet early warning satellites code-named Oko had orders to, in the detection of incoming missiles, start an immediate and compulsory counterattack against the United States.

On 26 September, the bunker's computers detected one intercontinental ballistic missile heading towards the USSR from the USA. Petrov's responsibilities included observing the satellite early warning network and notifying his superiors of any impending nuclear missile attack against the Soviet Union.

A suspicion was raised in the lieutenant's mind that this occurrence must have been the result of a software malfunction (the system's reliability had already been questioned in the past) since a first strike from the US was likely to involve hundreds of simultaneous missile launches in order to disable the Soviet Union's means of counterattack, so he decided not to follow his obligations and not retaliate, avoiding World War III. Later that day the computer detected four more incoming missiles that he dismissed as false alarms once again despite having no direct means to confirm this.

The Bug and its solving

It was subsequently determined that the false alarms were caused by a bug in the Soviet software that failed to filter out false missile detections caused by sunlight reflecting off cloud-tops, an error later corrected by cross-referencing a geostationary satellite.



Aftermath

Petrov first was praised for his actions and promised a reward, but later underwent intense questioning by the Soviets to justify his actions.

In the end, he received no reward, probably (as Petrov speculates) because the officials and the scientists in charge of the system were embarrassed by the incident and other bugs later found and if he had been commended, they would've had to be punished severely. The story was buried having only surfaced in 1990 due to the publication of Votintsev's memoirs.

Petrov was later commended and praised in the USA by several associations having won the Dresden Preis award in Germany, 2013.

Lessons Learned

When a piece of software is created, every possible case needs to be covered to avoid events like these. Software is part of our lives and will be with the passing of time, even more. Some say it will hold our lives in the future, in this case, I think it did because a lot of people could've died because of this problem and the world as we know it today, wouldn't exist. Everyone forgets something and there is no combination of tools to consider every possible case regarding a program, only the human mind, which we know can fail us sometimes and be the changing factor between a faulty code that can start a world war and a code that protects a country and its inhabitants from outside attacks.

References

[1] Marshall Brain. *How the Year 2000 Problem Worked*. URL. Apr. 2000.

[2] Dennis Burke. *All Circuits are Busy Now*. URL. Nov. 1995.

[3] The Editors of Encyclopædia Britannica. *Y2K bug*. URL. Dec. 2011.

[4] Peter G. Neumann. *Cause of AT& T Network Failure*. URL. Feb. 1990.

[5] Barnaby J. Feder. *Fear of the Year 2000 Bug Is a Problem, Too*. URL. Feb. 1999.

[6] Writers of National Geographic. *Y2K Bug*. URL. Jan. 2011.

[7] Christopher O'Malley. *When the Computer Fails*. Jan. 1990, pp. 84–90.

[8] Joel Spolsky. *The Law of Leaky Abstractions*. URL. Nov. 2002.

[9] Military Story's editorial team. *The day before: Stanislav Petrov and 1983 Soviet nuclear false alarm incident*. URL. Apr. 2016.

[10] Wikipedia. *1983 Soviet nuclear false alarm incident*. URL.

[11] Wikipedia. *Year 2000 problem*. URL.