Bugs

Adrian Kosmaczewski

2021-01-15

Somewhere between the end of the Second World War and the beginning of the Cold War, when the enemy started speaking Russian instead of German, a US Navy programmer was working in an early computer, trying feverishly to solve a problem in a program.

- Software is mostly bugs
- No guarantees
- Compile time vs. Run time
- An analogy
- Learning

Finally, says the legend, Mrs. Grace Hopper opened the computer chassis and discovered, between the cables and the bolts, a dead cockroach.

A bug, found during the first debugging session ever in the history of mankind.

This anecdote is illustrative in many ways, mainly because it contains almost no mention of the program that did not work – of course, its contents might have been considered state secret or something like that, but the truth is that the "bug" was actually a foreign body, completely unrelated to the program being executed.

Software is mostly bugs

In your professional life, however, most of the code you will write and use is buggy. Most of the libraries that you will download and use in your web pages contain bugs of very different kinds. Actually, working software will be the exception to the rule; most of the software does not work, and will never truly work all the time.

Why this is so you ask? Well, to begin with, a program is a promise that you cannot keep. You describe a world, a sequence of steps, a structure in memory that must be operated upon following a certain way and in a certain order. But the world is imperfect; maybe the computer running your software 10 years from now will have new security mechanisms that will block the system calls you are using; maybe it will use a certain type of memory chips that do not allow certain kinds of allocations; maybe there will not be enough memory at all.

But you, in your code, you are programming your application against an imaginary piece of hardware. You are going to be making tons and tons of assumptions, and you know what? That is ok. Everybody does that. The difference between a junior and a senior developer is the decreasing amount of assumptions made in one's head.

No guarantees

So, the thing is, no matter which programming language you choose, there are no guarantees.

Yet, in spite of all of the evidence, many developers feel a warm fuzzy feeling when using statically typed languages. It is like if they felt more protected; you know, this variable is *always* going to be a pointer to a string, or this other variable is *always* going to be an integer. They put up with ridiculously long compilation times, complex syntaxes, casting objects up and down their class hierarchies in order to do things.

Compile time vs. Run time

While other developers, maybe more aware of the reality of the world, happily write and publish applications in languages such as Ruby, Python, JavaScript or Objective-C, knowing well that if something can go wrong, it *will* go wrong.

C++, Java, Swift and other languages scoff at the thought of developers keeping track of the objects at the end of the variables, and make sure that every method call, that every function and every parameter and every possible combination of templates and generics actually makes sense to an increasingly complex theorem, verified at every compilation.

Yet, in all of its glory, even these allmighty languages have to declare themselves lost at some point, and they implement vtable lookups in their objects anyway, because there is always some situation in which the resolution of the polymorphic method to be called cannot be determined at compile time, and we have to cross fingers and pray that everything will be OK at runtime.

An analogy

When I used to teach iOS programming to developers, I used a very simple analogy that always made me laugh.

Somehow I thought of C++ as an east-coast kind of language: uptight, control-freak, obsessed with detail and verification, and without any trust whatsoever. If C++ was a person, it would be a rich financial trader in Manhattan, with a nice suit and a nice car, and a perfectly controlled life around him.

A bit like American Psycho, if you see what I mean.

On the other hand, I used to describe Objective-C as a pure west-coast kind of language, a product of the 70's, a relaxed language that would make almost no verification whatsoever of your types, while at the same time it would be smoking pot and listening to Led Zeppelin out loud. Objective-C would be a 1971 hippie in the middle of San Francisco, protesting against the war in Vietnam and enjoying life as it comes.

A bit like American Pie, if you see what I mean.

Learning

What I recommend to you, faced with strongly and weakly, statically and dynamically typed languages, is that you start your career with a really strongly typed one. It will make your beginnings easier, as long as you can understand the error messages of your compiler – and believe me, they can be quite hard to understand sometimes.

But as you grow up, let's say 10, 15 years from now, move to more dynamic languages. Statically typed languages, maybe something like Go (not particularly C++ to be honest, buy why not) that will provide you with a simpler mental model to follow at the beginning. The compiler will basically take you by the hand and guide you until you have a working piece of software.

However, let your inner instinct guide you towards more relaxed languages as you move forward. Open up your mind, and enjoy the tremendous freedom offered by a language whose compiler does not stand up in your way.