Issue 038: Design By Contract

Graham Lee

November 1st, 2021

Welcome to the thirty-eighth issue of *De Programmatica Ipsum*, dedicated to the subject of *Design by Contract*. In this edition:

- Adrian explains the origins of Design by Contract and its relation to assertions[1] and exceptions.
- Graham discusses how to work with mutable state[2] in your code through Design by Contract.
- In the Library section[3], Adrian reviews "Facts and Fallacies of Software Engineering" by Robert L. Glass[4].

Enjoy this issue! Please subscribe to our free newsletter[5] to stay updated about new releases, share the articles on social media, or contribute[6] if you would like to support our work.

Cover photo by Romain Dancre[7] on Unsplash[8].

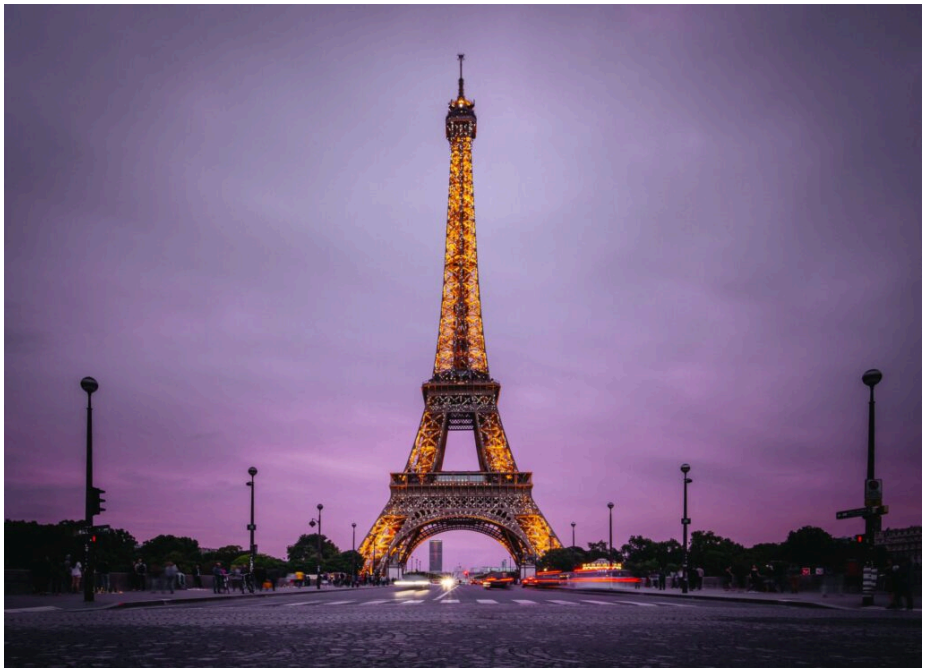
---

[1] https://deprogrammaticaipsum.com/assertions/
[2] https://deprogrammaticaipsum.com/how-to-reason-about-mutable-state/
[3] https://deprogrammaticaipsum.com/category/library/
[4] https://deprogrammaticaipsum.com/robert-l-glass/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://deprogrammaticaipsum.com/contribute/
[7] https://unsplash.com/@romaindancre?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[8] https://unsplash.com/s/photos/contracts?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Assertions

Adrian Kosmaczewski

November 1st, 2021



Colin Powell[1] passed away during the preparation of this article. In one of the most memorable and abhorrent chapters of his career he gave a speech in the United Nations trying to justify the unjustifiable. In that occasion, he said[2]:

> My colleagues, every statement I make today is backed up by sources, solid sources. These are not assertions.

Of course, neither the sources were solid, nor the statements were true. But that is another problem.

As a practitioner in the computer field, this author cannot but to feel baffled by this (mis)use of the word "assertion". Are not assertions, precisely, strong and definite statements? At least dictionaries say so[3], and in your code, they very much seem so too, to the extent that they will crash your whole program irrevocably should they ever fail. The world of software coopts words (such as `exception`, `static`, `public`) and sometimes gives them a completely different meaning; but in this case I think it was Mr. Powell who chose a more fitting meaning for his case.

The Word

What are assertions to computer programmers, then? According to Bertrand Meyer[4],

[1] https://en.wikipedia.org/wiki/Colin_Powell
[2] https://2001-2009.state.gov/secretary/former/powell/remarks/2003/17300.htm
[3] https://www.merriam-webster.com/dictionary/assert
[4] https://deprogrammaticaipsum.com/bertrand-meyer/

> An assertion is an expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution.

("Object Oriented Software Construction, Second Edition", 1997, page 337)

As explained in chapter 11 of Meyer's book, assertions are meant to check the *correctness* of a piece of software; that is, its ability to perform the tasks defined in their specification.

Because, you do have a specification, right? Right?

On the other hand, exceptions (a closely related concept to assertions, but not quite the same[5]) are meant check its *robustness*, that is, its ability to withstand abnormal conditions.

In section B6 of the "K&R" book, Brian Kernighan[6] and Dennis Ritchie explain that

> The `assert` macro is used to add diagnostics to programs.

("The C Programming Language, Second Edition", 1988, page 253)

Bjarne Stroustrup describes assertions while discussing exception handling in his *magnum opus*:

> A variety of techniques are used to express checks of desired conditions and invariants. When we want to be neutral about the logical reason for the check, we typically use the word *assertion*, often abbreviated to an *assert*. An assertion is simply a logical expression that is assumed to be **true**.

("The C++ Programming Language, Fourth Edition", 2013, page 360)

Stroustrup sets a very interesting point about the distinction between compile-time and run-time assertions, to which we will come back shortly before the end of this article.

Steve McConnell[7] gives a similar definition of assertion to Meyer's in "Code Complete":

> An assertion is code that's used during development–usually a routine or macro–that allows a program to check itself as it runs.

("Code Complete, Second Edition", 2004, page 191-192)

Among the various guidelines McConnell provides for assertions, the following two stand out:

- Use error-handling code for conditions you expect to occur; use assertions for conditions that should never occur.
- Use assertions to document and verify preconditions and postconditions. Preconditions and postconditions are part of an approach to program design and development known as "design by contract" (Meyer 1997)

Here we find the connection between assertions and the subject of this month's issue: assertions can be used to implement "Design by Contract" in software.

The phrase "Design by Contract" means many different things. First of all, it is a trademark[8], and this author hopes that our use of the phrase in this edition of the magazine will fall under the "Fair Use" clause of copyright law. Henceforth we will refer to it throughout this text using the proper casing defined by said trademark.

[5] https://chrisoldwood.blogspot.com/2016/06/confusing-assert-and-throw.html
[6] https://deprogrammaticaipsum.com/brian-kernighan/
[7] https://deprogrammaticaipsum.com/steve-mcconnell/
[8] https://tsdr.uspto.gov/#caseNumber=78342308&caseType=SERIAL_NO&searchType=statusSearch

Second of all, Design by Contract is actually more than just asserting. But, alas, in many programming languages asserting is a poor man's approach (sometimes the only one available) to implement a Design by Contract strategy. Some unlucky folks might even have to create their own assertion macro or function from scratch, but hopefully not anymore these days.

The Implementation

Some programming languages provide full Design by Contract features, usually provided through some library or framework: let us mention Python's deal[9] and PHP's deal[10] libraries; Rust's contracts crate[11]; and C++'s Boost.Contract[12] and a proposal[13] from 2017 to integrate it into the language.

Furthermore, Android developers will be happy to learn that Kotlin has Design by Contract baked in since version 1.3[14] introduced in 2018. The syntax is quite straightforward:

```
fun String?.isNullOrEmpty(): Boolean {
  contract {
    returns(false) implies (this@isNullOrEmpty != null)
  }
  return this == null || isEmpty()
}
```

Similarly, the `defer` keyword in Swift could be used[15] to provide some sort of postcondition contract checking.

The .NET Framework used to have code contracts[16] until recently, but they unfortunately never made it into .NET Core[17]. But Microsoft has bigger problems[18] with open source, anyway.

Finally, in Java one could use AspectJ[19] to "inject" pre- and postcondition checks through join points and annotations.

The Language

The quintessential programming language that brought Design by Contract to the spotlight was Eiffel[20], created by the aforementioned Bertrand Meyer.

In Eiffel, the implementation of Design by Contract explicitly specifies keywords and placeholders in methods, where the contract verification code must be placed. As an example, this quote from the book (page 349) should suffice:

```
class STACK2 [G] creation
  make
```

[9] https://github.com/life4/deal
[10] https://github.com/php-deal/framework
[11] https://crates.io/crates/contracts
[12] https://www.boost.org/doc/libs/master/libs/contract/doc/html/index.html
[13] https://isocpp.org/files/papers/p0542r0.html
[14] https://kotlinlang.org/docs/whatsnew13.html#contracts
[15] https://www.hackingwithswift.com/new-syntax-swift-2-defer
[16] https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts
[17] https://github.com/dotnet/docs/issues/6361
[18] https://dusted.codes/can-we-trust-microsoft-with-open-source
[19] https://www.eclipse.org/aspectj/
[20] https://en.wikipedia.org/wiki/Eiffel_(programming_language)

```
feature -- Initialization
  make (n: INTEGER} is
    -- Allocate stack for a maximum of n elements
  require
    positive_capacity: n >= 0
  do
    capacity := n
    !! representation.make(I, capacity)
  ensure
    capacity_set: capacity = n
    array_allocated: representation /= Void
    stack_empty: empty
  end
```

The code above, describing the well-known stack data structure, is quite self-explanatory, even if you have never seen a line of Eiffel before. The `require` block checks for the preconditions of the code, before the main body of the function (`do`) runs: we need this check, because even though the `STACK2` class constructor takes an `INTEGER` value, we cannot accept (for obvious reasons) a negative value. Should that ever happen, the runtime would announce a breach of the `positive_capacity` contract, and loudly so.

Inversely, the `ensure` block checks for the state of the newly created instance before returning control to the caller. Should the library code fail to create the underlying backing array, in a memory-constrained environment for example, the `array_allocated` contract would be broken, and the program would halt.

Design by Contract, as implemented in Eiffel and in some of the libraries enumerated above, goes a bit further away: not only are pre- and postconditions checked in methods, but also invariants can be set at class level. These contracts make it evident, at every single moment of the execution of the software, what the state of the component should be. Needless to say, this provides a higher level of correctness verification to software; even if a method fails to check correctly for parameters or state, the class itself is aware of it at any single time.

The Value

The writer of these lines assumes, not without a certain sadness, that most of our readers are not using Eiffel to build their microservice, full-stack, or mobile apps. Whatever the environment or runtime your code runs into, it is impossible to stress enough the importance of (ab)using the provided assertion mechanism by your language in your code.

Doing so will definitely make you more productive; furthermore, many languages allow you to disable those assertions from production builds should you wish to, allowing your code to run faster, but still providing help during writing, debugging, and testing.

Because, after all, those unit tests you cherish so much, they are simply assertions that you run from the outside, usually with a separate executable. Now, imagine sprinkling your code with those same assertions, but inline, all over the place, and having them executed at every step of their way. Such is the underused power of Design by Contract, through a simple mechanism that is freakishly accessible to most of us.

Bertrand Meyer summarized the benefits of Design by Contract in an interview:

> *How does Design by Contract help a team of developers?*

> Bertrand: It makes it possible for the various parts of the team to know *what* their partners are doing without having to know *how* they are doing it. (...) It is also very good for managers.

("Masterminds of Programming", 2009, page 422)

The Future

But is Design by Contract still relevant in the age of pervasive type inference[21] and rewriting in Rust[22]? Should not compilers provide all of these checks? Could not the `STACK2` Eiffel class above just use an `unsigned int` as a parameter for the constructor instead of using a contract?

As powerful as type systems are these days, to answer that question this author yields to Joel Spolsky, who told 20 years ago the story[23] about a seemingly impossible condition in C++ code: a `NULL` reference and its related crash. Hopefully the debugging of such situation did not take long, but an `assert()` would have certainly helped. Even though, to be honest, it might have seemed bafflingly useless to most reviewers; after all, the C++ specification explicitely excludes such condition.

Somewhat in this direction, it is worth mentioning Go's type assertions[24], or C++'s `static_assert` mechanism[25] introduced in C++11.

The need for reliable "mass produced software components"[26], as proposed by Douglas McIlroy[27] in 1968, requires, in the opinion of this author, both static typing *and* runtime checks. Statically compiled code cannot withstand the ever-changing conditions of all the machines where it might run, simply because, by design, the future cannot be completely foreseen. Design by Contract and assertions were, are, and will continue to be one of our strongest allies in this quest for reuse, composability, and quality.

After all, Mr. McIlroy invented Unix pipes, and his idea of software components was hailed as a potential "silver bullet"[28] by Fred Brooks[29] himself. McIlroy foresaw a thing or two about what the future of software components should be, and our world of ever changing, never API stable, `npm install` style of components is not it[30].

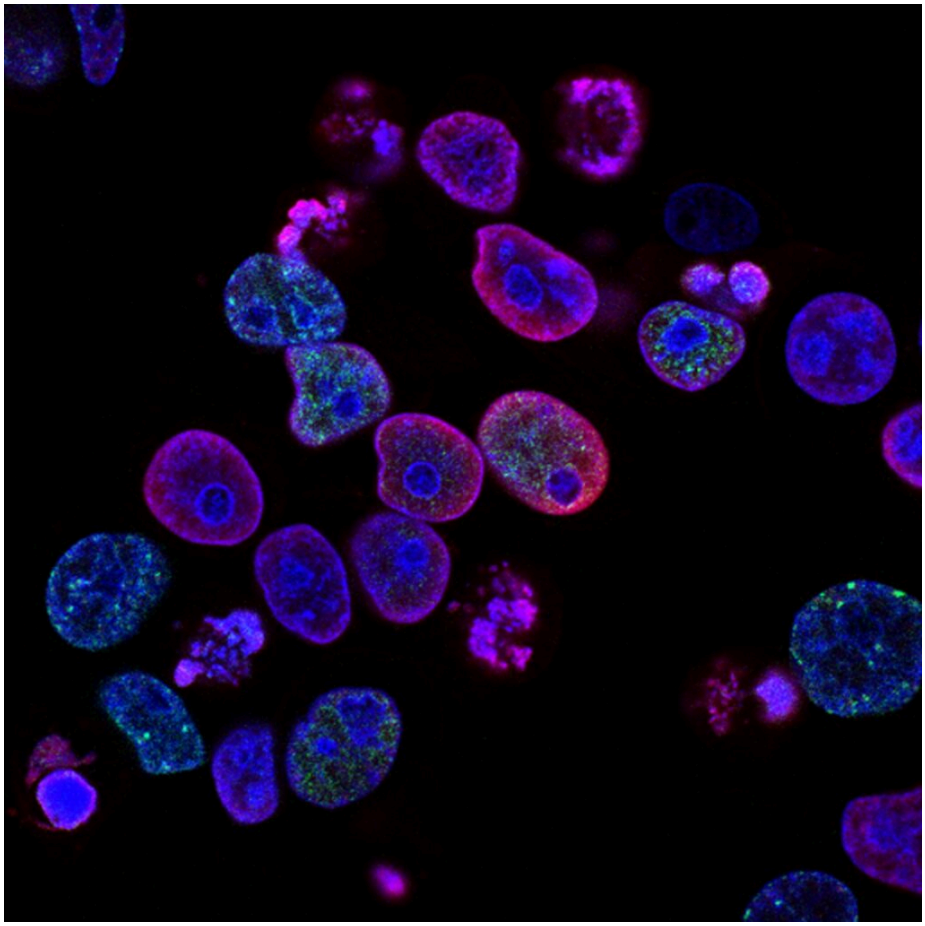
Cover photo by Denys Nevozhai[31] on Unsplash[32].

[21] https://deprogrammaticaipsum.com/the-truce-of-type-inference/
[22] https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/
[23] https://www.joelonsoftware.com/2001/07/31/hard-assed-bug-fixin/
[24] https://golangdocs.com/type-assertions-in-golang
[25] https://en.cppreference.com/w/cpp/language/static_assert
[26] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF
[27] https://en.wikipedia.org/wiki/Douglas_McIlroy
[28] http://worrydream.com/refs/Brooks-NoSilverBullet.pdf
[29] https://en.wikipedia.org/wiki/Fred_Brooks
[30] https://drewdevault.com/2021/10/17/Reliability.html
[31] https://unsplash.com/@dnevozhai?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[32] https://unsplash.com/s/photos/eiffel-tower?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

How To Reason About Mutable State

Graham Lee

November 1st, 2021



The idea, prevalent among those who would prefer you to use functional programming languages, that it is impossible to reason using non-functional programming languages due to the existence of mutable state, is newer than many of the reasoning mechanisms and tools that we use to understand programs that use mutable state.

Let us first unpack the idea of a "functional programming language", because I am sorry to tell you that there is no such thing. Wikipedia would give you a different impression: it would tell you that programming paradigms are an ontology of programming languages[1]; a categorisation system that lets us say that this language supports this way of writing programs, and that language supports that way of writing.

This clearly does not hold. One of the mathematical statements proved about computer programs even before there were any programmable computers is that any model of computer has, other than physical limitations like size and speed, the capability to run any program

[1] https://en.wikipedia.org/wiki/Programming_paradigm

written for any other computer. The reason is that any one of these computers can run a program that makes it work like any other one of these computers, then read the program written for the other computer.

Say I have a program for the (virtual) Haskell computer, for example the model of computation exposed by the Haskell programming language. Can I run it on the (equally virtual) Python computer? According to mathematicians Kurt Gödel, Alonzo Church, and Alan Turing, the answer is yes: I can write a Python program that runs Haskell programs (an interpreter) and I can run my Haskell program on that interpreter. When I do so, the Haskell program is running on the Python computer.

So which programming paradigm is Python a member of? The argument above tells us that it must belong to the same paradigm as Haskell. And it also must belong to the same paradigm as Smalltalk[2], Java[3], C, Logo, Eiffel[4], Intercal, COBOL, Algol, APL, Postscript, Raku...

If we want to define a programming language's "paradigm", we are going to need a fuzzier, more psychological definition of that language's paradigm. The programming language makes it easy to express ideas that were formulated using a particular mental toolset. This shows us the underlying truth, wooly though it is: the programmer is thinking about their program in a particular way, and expressing that thought through the medium of the programming language. That means the "paradigm" is actually the mental framework, which is the same meaning of "paradigm" that everybody else who is not a programmer uses. That is beneficial, programmers using words the same way that everybody else does would help in a lot of situations!

Now we could also notice that the paradigms (the thought models) themselves have interchangeable power[5], which is unsurprising because we are all thinking about programs that we are going to run on interchangeable computers. But we can stop here: what we need to bear in mind is that the paradigm is how we think about the program we are writing. Getting back to our starting point, it is really not true at all that if you are thinking about a program using mutable state, you cannot think about what your program will do.

Here is how it works. Your computer starts in some initial configuration, which we will call the environment, E. You do not have to think of this as the initial state of all the bytes in memory and all the buses and all the CPU registers, because your programming language provides some abstraction over this in terms of named slots or variables. The environment can be the values of the variables in the program, and an idea of what operation will run next. Again, the "what operation" does not have to be a machine instruction that is currently referenced in the instruction pointer or program counter register, it can be a programming language statement or a group of statements like a subroutine, procedure, function, or method.

So the computer runs the operation that should run next, which may take one clock cycle (it almost never does on a modern CPU) or many seconds depending on the complexity of the operation that is coming up. That is not important though, we pretend that the computer atomically "ticks" from E to some new state, E', which is the values of the variables and the instruction pointer *after* that operation has completed. For example, if it is the assignment statement x = 3; then E' is the same as E except for two things: x now has the value 3, and the instruction pointer has moved to the operation after the assignment. If the operation

[2]https://deprogrammaticaipsum.com/issue-25-smalltalk/
[3]https://deprogrammaticaipsum.com/issue-24-java/
[4]https://deprogrammaticaipsum.com/assertions/
[5]https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

is the go to statement, then the instruction pointer becomes equal to the parameter of the statement and the environment is otherwise unchanged.

These "environments", the before and after state, can become quite large (though we will see tricks to handle that later in the article), so rather than writing them exactly, we just think about *predicates*, boolean tests of the environment, and say that we will test the initial environment with one predicate, *p*, and the resulting environment with another, *q*. We can then write an assertion about the result of running a statement or subroutine S in the form *{p}*S*{q}*. That is: if *p* is true, and we run S, then *q* will be true.

This is called a Hoare triple[6] after C.A.R. "Tony" Hoare[7], and if you have written any form of automated software test then you are already familiar with the Hoare triple. A unit test is "assemble-act-assert": in other words, get into a state where *p*, run S, test *q*. A BDD spec or user story is "given-when-then": given *p*, when S happens, then *q*. The only real difference is that tests specify single examples, where as the predicates in Hoare triples can be general statements about the state of the computer: preconditions that must be true before S to ensure that the postconditions will be true after S.

As I said, the whole environment of the computer, even in the abstract model of a programming language, can be too much to think about, so we have a few tricks to deal with that. The first is structured programming. Dikstra and others explored the composability of these Hoare triples (for example, what happens when you run multiple statements sequentially) and showed how you get from sequences of statements about individual operations to preconditions and postconditions about whole loops, conditionals, and procedures. If you know the behaviour of the instructions in a function, you can gather the initial conditions of those together into a single precondition that must be true before the whole function is run. If you know the outcomes of the instructions, you can gather those together into a single postcondition that will be true after the whole function is run. Now wherever you want to call that function, you do not need to worry about the states of all the variables in the computer: you just need to ensure that the function precondition is true, and that what you do after is compatible with the postcondition being true.

This whole precondition-action-postcondition thing being tedious to write, we call it a *contract* after the similar real-world idea. A contract says "if you do this for me, then I will do that for you" and that is what it does in software too: if you make sure my preconditions are met, then I will guarantee to satisfy these postconditions.

Object-oriented programming takes this idea of design by contract even further, which is true in *all* OOP contexts even if it is only made explicit in the Eiffel programming language[8]. Instead of thinking about whole computers, we use encapsulation to break the world up into tiny sub-computers with small environments, and call those tiny sub-computers "objects". Inside this object, its instance variables are contextually relevant and everything else is out of scope. It will receive some additional parameters via messages, and can send information out via messages, but the only things it needs to track are its ivars. Outside the object, its ivars are off-limits and you can think purely in terms of the contract it exposes.

There is one trick left we need to know about. When we have concurrent threads of execution and shared memory, we cannot rely on arbitrarily bundling up larger and larger operations into atomic changes in environment and pretending nothing else happened in between.

[6] http://cs.iit.edu/~cs536/handout/class/c08.pdf
[7] https://en.wikipedia.org/wiki/Tony_Hoare
[8] https://en.wikipedia.org/wiki/Eiffel_(programming_language)

That point "and shared memory" is an important one though; with judicious design using encapsulation, you absolutely *can* treat methods as atomic actions, though you still have to wonder what is going on in the context where all the methods' effects are being applied. For those times, you have to think about all the different interleavings of operations on different threads and the effects of those sequences on the overall outcome. Is that it, then, is that the death of Design by Contract?

No, that is the time to reach for a computer. Remember those? We use them to render other people unemployed and give them fake news about whose fault it is, all the while pretending they are good tools for enhancing human thought. Well they actually *can* be used to augment thought, and you can use a computer to enumerate those interleavings and tell you whether a contract ever gets broken. When the functional programmers tell you they do not need to do that, remember to ask whether they or their compiler are doing all the type checking.[9]

Footnote: of course Bertrand Meyer and the folks at Eiffel took a different approach to concurrency[10]: rather than showing whether the contract can be violated for a given specification, they use the contracts to construct a (concurrent) program where violation is impossible.

Cover photo by National Cancer Institute[11] on Unsplash[12].

[9] https://lamport.azurewebsites.net/tla/tla.html

[10] https://www.eiffel.org/doc/solutions/Concurrent_programming_with_SCOOP

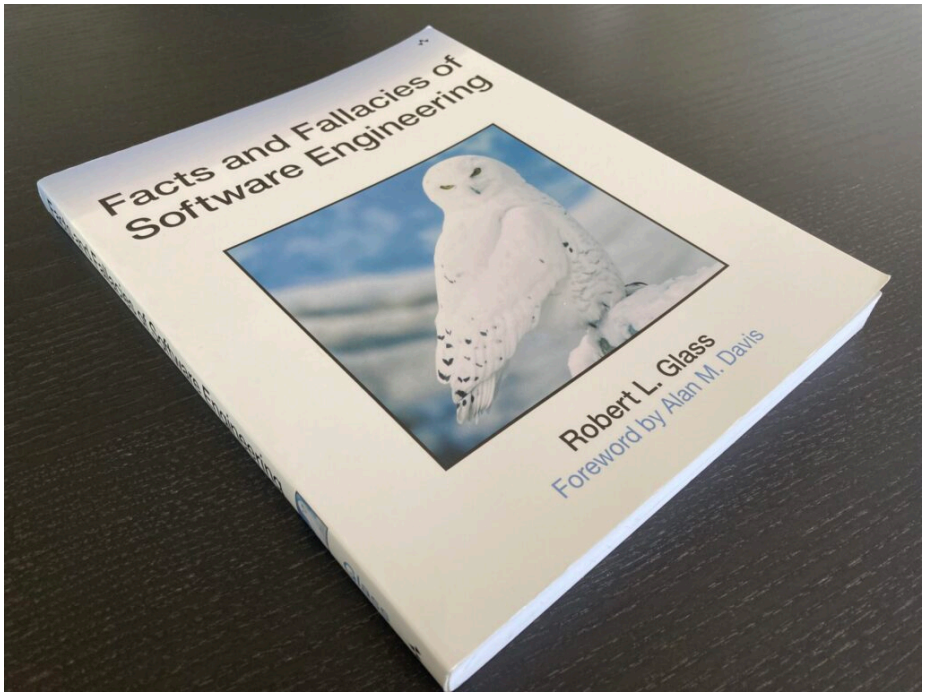
[11] https://unsplash.com/@nci?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[12] https://unsplash.com/s/photos/cells?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Robert L. Glass

Adrian Kosmaczewski

November 1st, 2021



Robert L. Glass[1] wrote a book in 1998 called "Software 2020"[2], currently rated with only one star... by the author himself. In his review[3] (written in 2017) he justifies this abysmal record because of a simple observation: none of the predictions in the book turned out to become a reality.

On the other hand, "Facts and Fallacies of Software Engineering"[4], published in 2003, has four stars out of five on Amazon. This opus, then, clearly outperforms the previous title by the same author. Maybe this success is due to the fact that, unlike its predecessor, this book does not predict the future; instead, it describes an eternally grim present, one that never seems to go away, no matter how many new versions of your debugger you try.

Because, let us be honest. Reading this book is pure masochism. Not because it is badly written (far from that) or that the contents are not relevant. Quite the opposite, actually; the sad truth of our craft is described in painful detail in every single section. Reading it is an exercise in nodding.

[1] https://en.wikipedia.org/wiki/Robert_L._Glass

[2] https://www.amazon.com/exec/obidos/ASIN/0942337050/developerdots-20

[3] https://www.amazon.com/gp/customer-reviews/R1TEBKSHE41A50/ref=cm_cr_dp_d_rvw_ttl?ie=UTF8&ASIN=0942337050

[4] https://www.pearson.com/us/higher-education/program/Glass-Facts-and-Fallacies-of-Software-Engineering/PGM247272.html

This book has received its fair share of reviews[5], slashdot threads[6], and discussions[7]. No need to indulge the reader into yet another discussion of the relative merits of each section, or whether or not the author agrees with each item. That is utterly irrelevant.

The core question raised by this book, re-reading it in *Anno Domini* 2021, is why is our industry still falling into the same traps? Why do managers keep considering lines of code as a valuable metric (fallacy 6)? Why is it that companies need to continuously break[8] backwards compatibility in their SDKs and IDEs (fact 6)? Why are schools not properly training younger developers to the reality they will face during their careers (fact 48 and fallacy 10)? Why is it that full-stack developers feel an urge to dive every year into the newly hyped[9] JavaScript framework *du jour* (fact 5)? Why are not product owners aware of the increase in complexity of every new feature added to the backlog (facts 21 and 23)? Why do some companies insist in 100% test coverage, even though it is not useful nor enough (fact 33)? Why do we keep on insisting on cramming developers in open space offices (facts 4 and 13)? Why do many teams still not adopt healthy code review practices (fact 37)?

Why is it that our craft is such a disgusting mess of appalling whimsical cargo-cult practices, disguised as perfectly coherent and rational ideas? A deep sigh ensues.

Not being able to resist the temptation, this author will briefly mention one fact and one fallacy to start reading:

- Fact 30: COBOL is a very bad language, but all the others (for business data processing) are so much worse.
- Fallacy 8: "Given enough eyeballs, all bugs are shallow."

This book can be read in a fully non-linear fashion, so feel free to explore this classic starting with these two. Think of this book as a guide, providing answers for most questions (almost all, I would say) you are going to have about your craft, your teams, and your code, for years to come.

As a personal wish, the author of these lines can only hope for a future in which our craft will work on each of these facts and fallacies, and work towards fixing them in the long run.

Cover photo by the author.

[5] https://blog.codinghorror.com/the-delusion-of-reuse/
[6] https://slashdot.org/story/04/08/27/1616256/facts-and-fallacies-of-software-engineering
[7] https://stalk-calvin.github.io/blog/2017/03/25/software-facts-fallacies.html
[8] https://deprogrammaticaipsum.com/issue-18-obsolescence/
[9] https://deprogrammaticaipsum.com/issue-1-hype/