Extended Abstract

Fatal Abstraction

Friedrich Steimann[a]

a Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, Germany

Abstract Abstraction is broadly considered a key asset in the making of software. However, the author finds that uncritical belief in abstraction puts software development at a substantial risk of failure. His essay combines some personal observations with more general concerns regarding the potential fatality of abstraction.

Keywords abstraction, programming language design, software engineering

The Art, Science, and Engineering of Programming

Submitted April 23, 2018

Published February 18, 2019

DOI 10.22152/programming-journal.org/2019/3/17

In *The Art, Science, and Engineering of Programming*, vol. 3, Essays, 2019, article 17; 4 pages.

This is an extended abstract of a paper that appeared in Onward! Essays 2018 [2].

The ability to abstract, and to reason in abstract terms, is one of the key achievements of the human mind. Without abstraction, there would not be language, and without language, there would not be software. As software people, we deal with abstractions all the time — in a computer there are no "real things," only abstractions, and if we are not good with abstractions, we had better choose a different profession.

Abstraction works in ordinary life because it enables us to ignore unimportant variations and details — a chair, for example, can be of many different forms, yet its function as a mechanism to avoid standing remains constant. Abstraction in software works because it enables us to ignore not only unimportant variations and details, but also to ignore implementation details, thereby reducing work and distractions. In both cases, the key ingredient is that abstraction enables is to ignore and thereby focus.

And yet, the use of abstraction is not without problems — problems solved or decisions made at an abstract level are useless unless they carry over to the concrete, where they are challenged by the details being abstracted from. One such detail may suffice to void an abstraction and everything that has been built on it. With details out of sight, such failures come as surprises, and the higher the towers of abstraction, the greater the risk of failure.

The essay "Fatal Abstraction" takes a look at the pitfalls of abstraction from three different perspectives: programming, software engineering, and management. For each it finds that in the best case, abstraction enables one to push back details temporarily but finds that the enthusiastic use of abstraction can sometimes defeat the benefits.

Abstraction in programming works best when the abstractions are tied to the domain the programs address. Such abstractions can be thought of as a domain-specific language (DSL). A DSL helps the programmer attend to the essential complexity of the problem while the abstractions hide the accidental complexities. Failures of abstraction can occur when a DSL is applied to other, perhaps related, problem domains, and the failures have to do with abstraction misfits and language bloat. A language with a number of embedded DSLs can require programmers to dig into the details being abstracted from, thereby diminishing their value.

Sometimes an abstraction cannot hide its details sufficiently — this is called a "leaky abstraction." In some cases the implementation of the abstraction is inadequate. For example, if a timer is documented to be precise to one-tenth of a second but is implemented as a binary float, testing for exact integer values may fail. This is similar to the problem of producing a decimal representation (a printed representation) of a binary float in which no information is lost, no garbage digits are produced, and the output is correctly rounded. In other cases, performance and other imperfections leak through.

Abstraction in software engineering can fail based on the *inherent flatness* of data as it has evolved over the years. It is common to construct a class hierarchy whose natural generalization (abstraction) is a superclass that does not represent the important details of its subclasses, and in the essay such examples are shown. This is a failure of abstraction related to the concept of non-monotonicity in knowledge representation —

it is a sort of overgeneralization common in top-down designs, though it is less common in bottom-up abstractions.

Nevertheless, it is typically possible to design tests that reveal errors of overgeneralization, but such tests rely on enumeration of cases.

> *The introduction of suitable abstractions is our only mental aid to reduce the appeal to enumeration, to organize and master complexity.*
>
> — Edsger W. Dijkstra [1]

Managers rely on abstractions they did not create. Therefore, projects led by managers generally proceed top down. In principle, the failures of abstraction in management mirror those in software engineering: as an idea emerges, more details become apparent, which may require a change in the abstractions — this in turn may challenge decisions based on previous abstractions. The difference is that managers who base their decisions on borrowed abstractions have to rely on others to sense abstraction failures, and to replace abstractions when this becomes inevitable. However, abstractions typically fail when confronted by the concrete, and so those who uncover abstraction failures cannot adequately communicate the problems to managers, because managers are familiar only with the abstractions that hide those very offending details.

To avoid abstraction failures, a principle is proposed called *the monotonicity of abstraction*, similar to monotonicity in reasoning. If **A** is an abstraction of **B**, then **A** is monotonic with respect to **B** if no decision based on **B** contradicts a decision based on **A**. That is, no decisions based on **A** need to be revised as more details about **B** are revealed.

Monotonicity of abstraction can lead to redundancy, but such redundancy can have beneficial consequences.

Abstraction is considered the holy grail of programming. To uncover flaws in abstraction might seem against the grain of common sense — perhaps it is instead the beginning of wisdom.

References

[1] Edsger W. Dijkstra. “Stepwise Program Construction”. In: *Selected Writings on Computing: A Personal Perspective* (1982). DOI: 10.1007/978-1-4612-5695-3.

[2] Fredrich Steimann. “Fatal Abstraction”. In: *Onward! 2018 Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Nov. 2018). DOI: 10.1145/3276954.3276966.

About the author

Friedrich Steimann can be contacted at steimann@acm.org.