Revisiting "Good" Software Design Principles To Shape Undone Computer Science Topics

Florence Maraninchi
Univ. Grenoble Alpes, CNRS, Grenoble INP[*], VERIMAG, 38000 Grenoble, France

1 Introduction

In a position paper last year [1] I suggested the notion of *anti-limits* to qualify all the cases in which the development of a digital system makes both a promise to the users, and an implicit design assumption, that some resource will grow as needed. I wrote: *Whatever our personal opinion, as computer scientists we can start exploring the notion of limits even if we do not agree on the moral judgments related to the choice of those limits. We can even explore the notion of limits without being convinced that there should be limits in the first place, just because this is a fascinating territory of undone science [2]. How to stay within* limits *has become a scientific and technical problem that it is little addressed.*

In this short paper, I will avoid the moral discussions and the sacrifices end-users might be asked to accept, in order to focus instead on the potential responsibility of the design principles that are considered *good* principles. My hypothesis is that they might well be an intrinsic obstacle to the emergence of research ideas for studying which type of digital systems would be acceptable in a sustainable world.

I will use as an example the hardware/software design principle on which most modern digital systems rely, namely *memory hierarchy* (caches, virtual memory, ...). My hypothesis is that the *transparency* of this mechanism and the induced design principles applied by software developers make it impossible to think of what it would mean to design digital systems within limits. Revisiting these principles is part of *undone computer science.*

2 The Illusion of Unlimited Fast Memory

In a very popular book [3] I use in my teaching activities, the introduction of the chapter dedicated to exploring memory hierarchy starts with the sentence: "*From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics we will look at in this chapter all focus on aiding programmers by creating the illusion of unlimited fast memory*".

First, let us explain why creating the illusion of unlimited fast memory indeed helps programmers. This is related to the general problem of sharing resources between various programs on a computer. Resources include memory and computing time but in the sequel we will focus on memory only. The solutions are all arranged on a continuum between two extreme points, detailed below.

[*]Institute of Engineering Univ. Grenoble Alpes

The Global View of a System Integrator

One of the extreme points is the following: in contexts where a *system integrator* exists, there exists a point in time when somebody knows all the programs that may ever run on the computer, and also precisely how much memory they will ever need. This resembles the situation faced by designers of critical embedded systems, for instance those found in trains or planes. The *system integrator* is the company which decides which hardware to buy, and which is also responsible if anything goes wrong.

The system integrator may decide once and for all, even before the programs are developed, which proportion of the memory will go to which program. The sharing can be done *statically.* Each developer team then works as if it had its own smaller computer, knowing the size of its share of available memory, and taking it as a non-negotiable constraint. The resulting systems are not meant to be extensible.

Open Systems

The other extreme point is when the system is essentially *open* to new programs or new versions of the already-installed programs that may need more memory. In this case you need your computer operating system to be able to share memory *dynamically.* But you also need to provide developers with an abstract interface to the actual memory of the computer, so that they can write their programs without knowing whether there is enough memory at some point of their execution, and where exactly their data will be stored. This is provided by the *transparent memory hierarchy principle.* As mentioned in the text cited above, with this abstract interface the operating system *creates the illusion of unlimited fast memory* for the developers. It makes sure that at any moment in time, the data needed by the running programs, at the execution point where they are, is present in fast memory. It does so by moving data back and forth between the fast memory and other types of memory. Globally, the total amount of memory needed by all the programs installed on a computer is far bigger than the actual amount of fast memory that is physically available. The system works because the programs are not all alive at the same time, and each of them does not need all its memory space all the time.

It is the typical situation for personal computers or smartphones, for which the end user is the actor who decides which programs to install. But if they try to install two many programs, and run them all at the same time, at some point the limit on the physical memory available will be reached anyway. The end user will be the one noticing the problem. The only solution for them will be to install more memory if it is technically possible, or to buy a more powerful smartphone. Reaching out to the developers of the applications is not going to solve the problem, because their design was done with the illusion of unlimited fast memory. It does not mean they did not care about memory use, because performance is an important economic criterion. But applications were not designed in a way that would allow for *shrinking their needs.* This operation would often be quite difficult without rethinking the architecture or the functionality entirely.

3 Closed Systems or *"Shrinkability"* as Unexplored Design Principles

Creating the illusion of unlimited fast memory has been dramatically effective in allowing developer teams to work independently of each other. In turn, independent design is probably the main reason why digital systems have reached the level of complexity we see today. This explains why it is almost universally considered a *good design principle.* Another design principle which is usually considered a *good* principle is *extensibility*, "a software engineering and systems design principle that provides for future growth", according to its Wikipedia definition. The difficulty to extend a design is considered a problem that needs a solution, see for instance the description of the factory pattern in [4]. The normal activity of a developer team maintaining a given application includes working on *extensions*.

But if we want computer systems to stay withing limits, what kind of principles would we need? This is where unexplored hypotheses abound.

One could for instance revisit modern computer systems by considering them as critical ones, asking for a system integrator to design the whole functionality beforehand. For a smartphone, it would mean that the end user is not allowed to install new applications, because the set of functionalities is predefined.

Another research direction would be to accept the notion of an open system, but to allow developers to make their applications *shrinkable* instead of *extensible*. We would teach how to write *shrinkable* instead of *extensible* code, and the normal activity of a developer team maintaining a given application would include the production of less and less-demanding versions, instead of versions with more and more functionalities. Instead of the end-user being faced with poor performances due to a large number of updated, hence more-demanding, applications running on their smartphone, applications could be updated with less-demanding versions. Considering the far-reaching consequences of such a paradigm shift in software development is also part of undone science.

References

[1] Florence Maraninchi. Let us not put all our eggs in one basket. *Commun. ACM*, 65(9):35–37, sep 2022.

[2] David J Hess. *Undone science: Social movements, mobilized publics, and industrial transitions.* MIT Press, 2016.

[3] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface.* 2nd edition, 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns.* Addison-Wesley, Longman, 1995.