Mètode Science Studies Journal
ISSN: 2174-3487
ISSN: 2174-9221
metodessj@uv.es
Universitat de València
España

Valverde, Sergi
The long and winding road: Accidents and tinkering in software standardization
Mètode Science Studies Journal, vol. 11, 2021, -, pp. 91-97
Universitat de València
Valencia, España

DOI: https://doi.org/10.7203/metode.11.16112

Disponible en: https://www.redalyc.org/articulo.oa?id=511766954008

MONOGRAPH
MÈTODE SCIENCE STUDIES JOURNAL, 11 (2021): 91–97. University of Valencia.
https://doi.org/10.7203/metode.11.16112
ISSN: 2174-3487. eISSN: 2174-9221.
Submitted: 18/11/2019. Approved: 10/03/2020.

THE LONG AND WINDING ROAD

Accidents and tinkering in software standardization

SERGI VALVERDE

Software is based on universal principles but not its development. Relating software to hardware is never automatic or easy. Attempts to optimize software production and drastically reduce their costs (like in hardware) have been very restricted. Instead, highly-skilled and experienced individuals are ultimately responsible for project success. The long and convoluted path towards useful and reliable software is often plagued by idiosyncratic accidents and emergent complexity. It was expected that software standardisation would remove these sources of unwanted diversity by aiming at controllable development processes, universal programming languages, and toolkits of reusable software components. However, limited adoption of development standards suggests that we still do not understand why software is so difficult to produce. Software standardisation has been limited by our poor understanding of humans' role at the origin of technological diversity.

Keywords: software standards, software development, programming language, complexity, evolution of technology.

Imagine that, when writing a love letter, you were forced to write sentences with a fixed number of characters. Forget about complex prose or mentioning Shakespeare in your masterpiece: if you go beyond the limits, you will have to stop your sentence whether it is finished or not. This happened to every student of computer programming in the 1980s and 1990s, including myself. At that time, it was necessary to break statements longer than 80 characters in smaller chunks to fit the limitations of text editors. This was even more noticeable when writing complex calculations in the respected programming language Fortran. Sometimes an equation could not be written in the space of a single line, and we had to split the long mathematical expression into multiple statements. Insertion of annoying line breaks in the middle of your thoughts seemed like an unjustified complication for a task – computer programming – that was (and still is) intrinsically complex. Why 80 and not 166 or any other number of characters?

The origin of this puzzle is earlier than fixed-size screens. The root of this accident is the size of punched cards used to process the US census in 1890s. These stacks of punched cards with 80 characters per line were fed to the first IBM commercial computers in the 1950s. Our personal computers inherited the 80-column format, which became the *de facto* standard known by many of us. Even today, when high-resolution displays are commonplace, text editors maintain compatibility with hardware relics that we will never use again.

«The exponential trend of hardware technology has not been mirrored by parallel improvements in software technology»

As exemplified above, historical accidents can leave deep fingerprints in the evolution of technology (Arthur, 1994). Some of these accidents can be harmless. But others generate inefficiencies associated to non-optimal tasks. How to guarantee optimality of technological choices?

One way is to check the collection of good practices and recommendations in the field. A technological standard removes unnecessary elements from engineering practices while preserving the «good stuff». These standards have saved a lot of effort by making sure that «materials, products, processes, and services are fit for their purpose», as the International Organisation for Standardization (known as ISO) declares. Many examples of useful standards are found in industry and engineering, where community of experts define and update their corpuses of solid recommendations. Crucially, the quality of these standards depends on experts' ability to decide when the norms and associated inventions are still useful or no longer interesting. Standardisation increases technological efficiency, but it can also prolong existing technologies to an excessive degree by inhibiting any investments in novel developments (Tassey, 1999). Reaching an optimal balance between efficiency and innovation is extremely difficult, and our predictions of technological innovations have been notoriously poor. In particular, complex innovations face higher obstacles for market success than simple ones (see Figure 1) (Schnaars & Wymbs, 2004).

Figure 1. Ever since the telephone was invented, its inventors predicted long-distance visual interactions. In 1924, Alexander Graham Bell said that «the day would come when the man at the telephone would be able to see the distant person to whom he was speaking». However, the attempted commercialization of the Picturephone by AT&T in the 1960s (see picture) was a market failure. AT&T invested so much in this technology that if commitment alone were the key to market success than the videophone should have been as commonplace as the telephone many years ago.

SOFTWARE BOTTLENECK

In the last century, we have witnessed a spectacular acceleration in computing performance, digital storage capacity, and world-wide electronic communications. The well-known Moore's law is the signature of the evolution of information technology (IT). This is the consequence of solid theoretical principles: the conceptual foundations of computers remain the same since the publication of Alan Turing's classic works. On the other hand, the exponential trend of hardware technology has not been mirrored by parallel improvements in software technology, which are still measured in human timescales. Although both hardware and software have grown in complexity, there is an important asymmetry in their evolution (Valverde, 2016). The current bottleneck in IT is not the cost of computer hardware, but obtaining the necessary software needed to run them (Ensmenger, 2010). Software dictates the utility of IT and the demand for software has created a huge economic problem. Many software projects are plagued by errors, accidents and idiosyncratic decisions (Brooks, 1975). The recurring project failures urged the community to define reliable approaches to high-quality software (Charette, 2005).

«Many software projects are plagued by errors, accidents and idiosyncratic decisions»

Ever since the invention of computer technology in the 1950s, standardisation has been an aspiring goal for associations of computer professionals and users. As an emergent field, computer professionals were eager to demonstrate their social utility. In particular, the programmers' initial standardisation efforts focused on software interoperability, i.e., the capacity to exchange and reuse software between different computers. For example, the standard operating system MS-DOS led to widespread adoption of personal computers (PC), which in turn allowed many subsequent innovations, like the emergence of the Internet and the World-Wide Web. The exponential growth of the computer market and the proliferation of incompatible computers increased the competition

for software. It was expected that standardisation would lead not only to compatible but also affordable software. To do so, standards focused in two main aspects: writing and maintain software code (or «software development») and the tools assisting in this process (e.g., programming languages and toolkits of reusable software components). But while software interoperability has been a great success, the limited adoption of software development standards suggests the presence of poorly understood constraints.

UNPREDICTABLE SOFTWARE DEVELOPMENT

Initiatives led by the US Department of Defense (DoD) are a good example of the obstacles faced by software development standards. From the 1970s to the 1980s, the DoD attempted to enforce software standards to their contractors (McDonald, 2010). The goal of the DoD was to reduce the huge costs of software development. Underlying this (and other parallel initiatives) has been the (never-reached) aspiration of replacing the human component by a fully-automated, error-free, process of software development. In 1978, the DoD published a series of software design rules that had to be followed by any software contractor: 1) software should be developed according to a top-down design process from the global system definition down to its functional parts (see Figure 2), 2) to improve the readability of software codes (and thus reducing the chances of error) there was a maximal size of individual software parts and the usage of «harmful» machine instructions (like "GO TO" instruction, which breaks the logical sequence of software operations) were forbidden, and 3) all software codes should be written using a listing of approved high-level programming languages. Surprisingly, the DoD faced much opposition when enforcing these rules to contractors. Although the standard reflected the conventions about well-written software, many programmers felt it was inadequate and obsolete, and an unnecessary burden limiting their freedom. Due to increasing criticisms and social pressure, by 1990s, the DoD abandoned any attempt of contractually imposing software standards.

A fundamental obstacle was the unrealistic assumption by the top-down model that the trajectory of software projects can be planned. Actual software (as well as many other complex engineering projects) often involves expensive fixes in latter stages of the project, that is, some of the initial design choices become historical accidents (McDonald, 2010). Experience with software projects suggests how difficult to meet functional requirements is, i.e., determining the set of tasks to be performed in software. In particular, any missing specification in the initial design translates to expensive modifications later in the project lifetime (Boehm, 1976). For example, users have an intuitive feeling about how operating systems (like Microsoft Windows) should work, but they have much difficulty describing software functions they have never used.

A more pragmatical approach conceptualizes software as an incremental and iterative process that is not very different from natural evolution. This hopes to minimize the amount of redundant design decisions. For example, in the prototype-based model, users and programmers actively cooperate when building a large and stable software system. Here, programmers' changes are a source of natural variation in the project. Users act as the environment for the software prototype by selecting features according to their needs. By repeating this iterative process, users and the programmers co-evolve a system that fits the specifications. Software prototyping agrees that, in a changing environment, quick and dirty adaptability is preferable over inadequate top-down planning.

«A more pragmatical approach conceptualizes software as an incremental and iterative process that is not very different from natural evolution»

AN ELECTRONIC TOWER OF BABEL

The tools used in software development have also not reached the uniformity of other technological fields, like electrical engineering. Technological diversity is found in the public repositories of open-source software, where there is not a single framework for developing software but many. Here, we can find many instances of the same problem solved in software for different platforms (e.g., Windows, Mac OSX or Linux), written in incompatible programming languages (e.g., C++, Python, or Java), and involving a mix of proprietary software libraries (e.g., OpenGL or DirectX). These solutions are based on the same ideas and concepts, but their underlying technologies are often incompatible, which limits their reusability. In this context, successful software integration still depends on the goodwill and voluntary

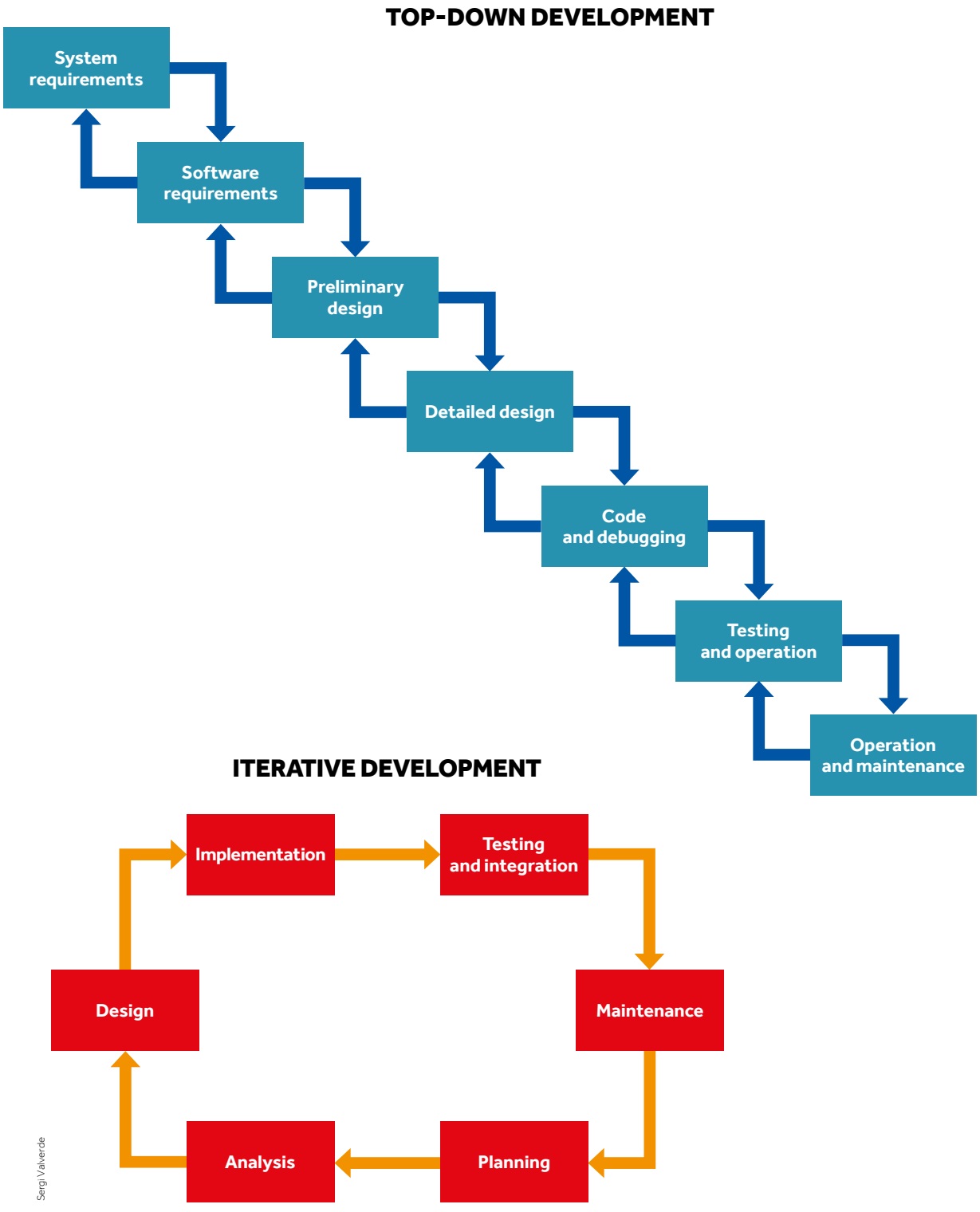


Figure 2. Standards divide software development work into different phases, such as design, build, test, and maintenance. In the 1960s, the Department of Defense of the US tried to impose a sequential (or top-down) model of software development to its contractors, with little success. Iterative software development is more flexible and can reduce misunderstandings between software users and programmers.

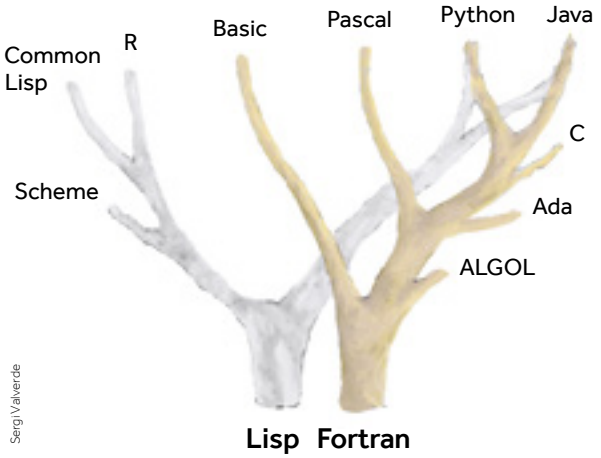


Figure 3. This schematic illustrates the branching evolution of programming languages. The target of early programming languages in the 1960s were mainly industry and businesses. Languages of this era, like ALGOL, were designed by private committees of experts. However, general adoption of information technology accelerated the diversification of programming languages. Popular languages like C or Python have been developed by distributed communities of software practitioners. Different branches influence ongoing evolution of programming languages.

collaboration among programmers. The situation of programming languages is particularly telling. In 1936, Alan Turing showed there are no theoretical barriers to the unification of programming languages, i.e., there is a universal computer capable of doing any task. However, the reality is that we have thousands of different programming languages at our disposal (see Figure 3). Why we cannot just create a universal language with many functionalities?

Standardisation of programming languages has been attempted many times, but none of them has been universally accepted. For example, the aim of the standard language Ada was to replace the myriad of programming languages used in DoD software projects (more than 450 languages were used in these Army projects by the 1970s). Unlike many languages designed by private committees (like COBOL), the design of Ada was the outcome of an international competition subject to external review (which included academic experts of programming languages). A design goal of Ada was to prevent human errors in software development, which features strict requirements of safety and concurrency rarely present in other languages. In spite of these benefits, Ada never gained the popularity of less robust languages like C++, which appeared in 1985. Some twenty years later, ISO standardised the language C++ for the first time (it is a *de facto* standard). Again, this suggest how success is not predictable no matter how much effort we spent in the initial design. Many factors influence the success of programming languages, including popularity, complexity and economics.

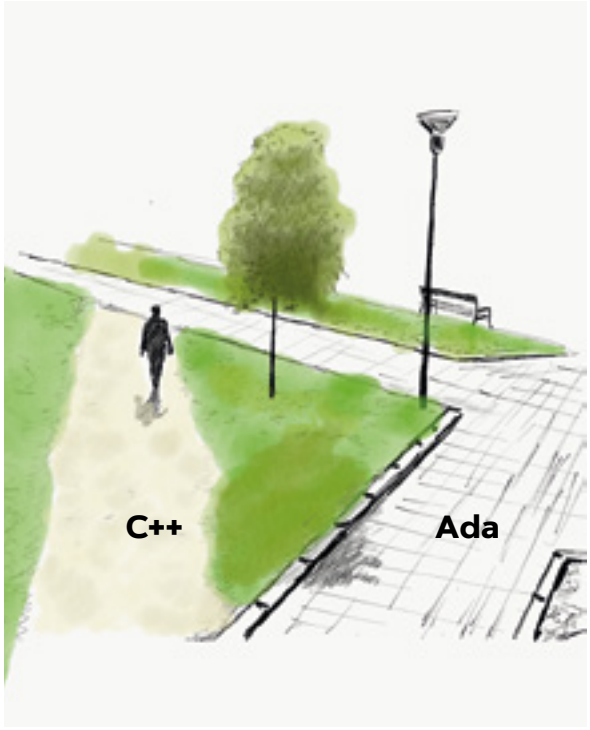
From a historical perspective, it seems the path from the language C to C++ was easier to follow than adopting a high-quality (but relatively unknown) standard (see Figure 4). Commitment to the «lesser» option did not translate in market lock-in or shortage of innovations because programming languages are continuously evolving and influencing each other. At some point, the large community of C++ users benefited from Ada innovations, while keeping compatibility with existing technology. It seems programmers prefer to live with imperfect software that to rebuild everything from scratch.

EMERGENT COMPLEXITY

In complex engineering processes, e.g., those involving software, engineers cannot always decide the best course of action. Instead, engineers are "driven" by the emergent complexity of their inventions. Both the evolution of technology and biology cannot avoid tinkering and accidents when complexity is very high.

In the early 1990s, the heterogeneous diversity of hardware, operating systems and programming languages was an obstacle for software interoperability. Engineers and managers saw component-based reuse as the natural solution to this problem. Instead of building software from scratch, an existing repository of building blocks (or components) of common software functions could be reused. This approach requires the definition of a software interoperability standard, such as CORBA (or Common Object Request Broker Architecture). In CORBA, software components written in different languages, e.g. Java and C, can still exchange information by adhering to a common software protocol. This standard was defined as the «next generation technology for e-commerce» and it gained a lot of popularity at the beginning. However, technical deficiencies and design flaws difficulted its market consolidation, and eventually CORBA was displaced by Web-related technologies, like XML (Henning, 2008).

«The tools used in software development have also not reached the uniformity of other technological fields, like electrical engineering»



Sergi Valverde

Figure 4. The path towards the adoption of technologies depends on many factors, including their cost. And the fact that some programming languages have been carefully designed by committees of experts does not automatically imply their widespread adoption. The standard language Ada (right path) is recognized as a high-quality language (with many unique features rarely present in other programming languages). However, the popularity of the operating system Unix catalysed adoption of its programming language C, and also its object-oriented successor, C++ (left path). These two programming languages –both standardised by ISO– have come to dominate the market while Ada has remained a niche solution.

The failure of CORBA has been mainly associated to quality issues. At a deeper lever, it exemplifies the difficult task of defining a standard interface between software components. Component-based software development is very similar to the idea of the Lego construction game. Lego bricks are interoperable thanks to the patented interlocking mechanism (see Figure 5). That is, we can connect any pair of bricks with independence of their shape, colour or function. This is not the case in software. Many historical examples teach us painful lessons about unwanted interactions between software components. To prevent this, we have been forced to test (and debug) any interaction, which takes a lot of time and effort. This is unavoidable and imposes a hard limit on scalable software development.

EVOLVING THE FUTURE

In a very short amount of time, we have transitioned from a science fiction-like view of all-powerful computers to a mundane commodity in the hands of everybody. This widespread adoption of information technology turned software into a key component of our society. There is a pressing need for more reliable and cheaper ways to develop software at faster rates. Software standardisation has been proposed as the solution to this problem, paralleling the thousands of essential standards needed to run our society.

Can we define universal rules to control software development? Recent history shows that reliable software development remains an elusive goal. A main problem is the uncertainty found at each stage of the software development process. When designing a software system, there are many different options. Deciding what is the best option at each stage is far from obvious. We cannot be sure about the long-term uses of technology because their success relies on unpredictable environmental changes. Uncertainty affects the evolution of many technologies (Petroski, 1992), but in software this is aggravated due to the absence of a physical embedding. Software does not decay or break in the same way as technologies do. Knowledge of the principles underlying physical systems allowed spectacular advances in engineering and industry. But the situation is quite different in software, where the scientific approach to software development currently lies behind practice. There is a need for maturation of software development, which depends on the availability of scientific results (Glass, 2009).

«Uncertainty affects the evolution of many technologies, but in software this is aggravated due to the absence of a physical embedding»

The obstacles to software standardisation suggest a recurring theme: human ingenuity cannot be replaced by standard parts. At the moment, brains are an essential component of translating human requirements to the computer language. We do not fully understand how humans program computers. Developing useful and reliable software requires ingenuity and considerable expertise. Inexpert programmers cannot rely on their intuition to assess whenever software is large or small, simple

Public domain

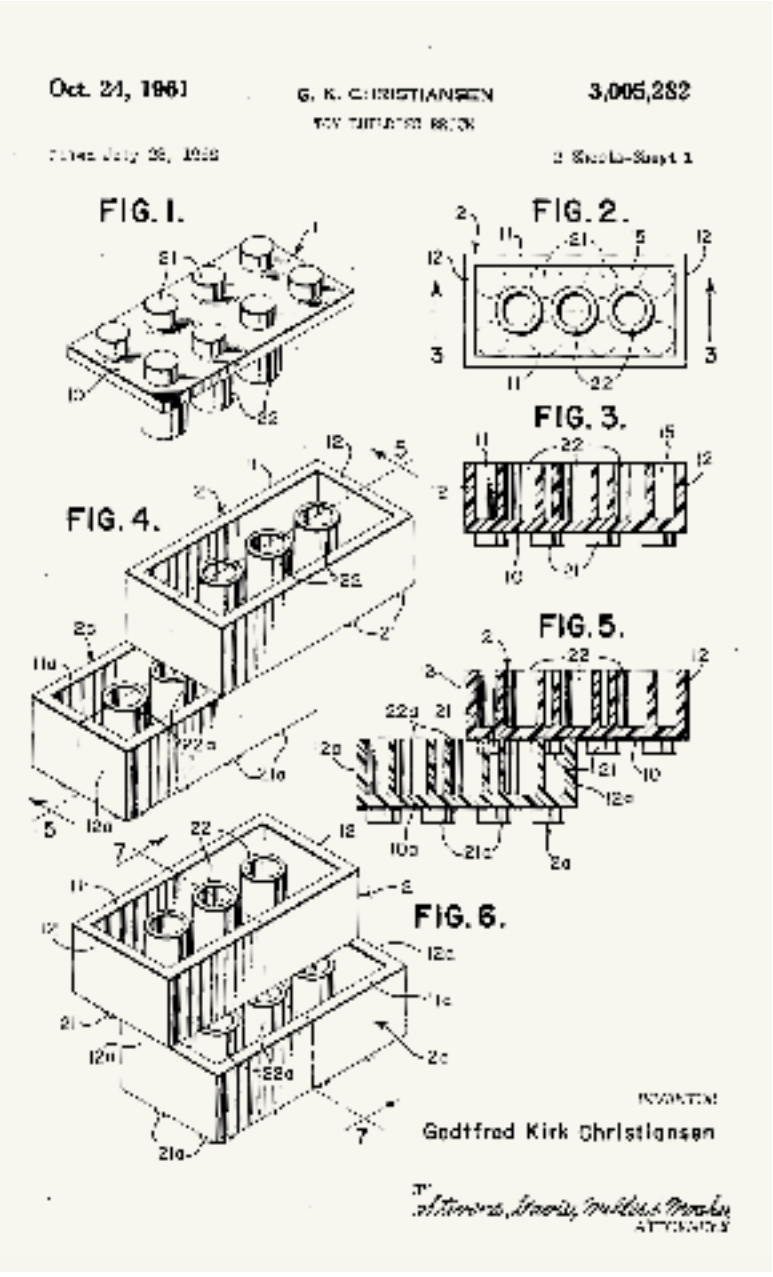


Figure 5. In 1961, US Patent 3005282A proposed a design for a «toy building brick», also known as LEGOs. This document describes an ingenious interlocking mechanism that allows many different toy structures to be assembled. A universal interface like this has never been achieved in software.

«We do not fully understand how humans program computers»

or complex, because it is invisible. Only when we spent a lot of time programming (and mostly debugging) computers, we start to grasp the sheer complexity of software. Beyond our intelligence and skills, we can only develop complex technology thanks to all the knowledge gathered by our society over the years (Basalla, 1988; Messoudi, 2011). The evolution of complex technologies like the programming language C++ has been the outcome of accumulating information by many people over many years in open-source communities. Some defend the idea that software development will be soon obsolete, and that artificial intelligence will replace entire communities of human programmers. This seems very unlikely without a full understanding of how humans program computers. In either case, one thing seems certain: software will not be designed, but evolved.

REFERENCES

Arthur, W. B. (1994). *Increasing returns and path dependence in the economy*. Michigan University Press. http://doi.org/10.3998/mpub.10029

Basalla, G. (1988). *The evolution of technology*. Cambridge University Press. http://doi.org/10.1017/CBO9781107049864

Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, *25*(12), 1226–1241. http://doi.org/10.1109/TC.1976.1674590

Brooks, F. (1975). *The mythical man-month: Essays on software engineering*. Addison-Wesley.

Charette, R. N. (2005, September 2). Why software fails. *IEEE Spectrum*. https://spectrum.ieee.org/computing/software/why-software-fails

Ensmenger, N. L. (2010). *The computer boys take over. Computers, programmers, and the politics of technical expertise*. The MIT Press.

Glass, R. L. (2009). Doubt and software standards. *IEEE Software*, *26*(5), 104. http://doi.org/10.1109/MS.2009.126

Henning, M. (2008). The rise and fall of CORBA. *Communications of the ACM*, *51*(8), 52–57. http://doi.org/10.1145/1378704.1378718

McDonald, C. (2010). From art form to engineering discipline? A history of US military software development standards, 1974-1998. *IEEE Annals of the History of Computing*, *32*(4), 32–47. http://doi.org/10.1109/MAHC.2009.58

Messoudi, A. (2011). *Cultural evolution: How Darwinian theory can explain human culture and synthesize the social sciences*. University of Chicago Press.

Petroski, H. (1992). *To engineer is human: The role of failure in successful design*. Vintage Books.

Schnaars, S., & Wymbs, C. (2004). On the persistence of lackluster demand: The history of the video telephone. *Technological Forecasting and Social Change*, *71*(3), 197–216. http://doi.org/10.1016/S0040-1625(02)00410-9

Tassey, G. (1999). Standardization in technology-based markets. *Research Policy, 29*(4-5), 587–602. http://doi.org/10.1016/S0048-7333(99)00091-8

Valverde, S. (2016). Major transitions in information technology. *Philosophical Transactions of the Royal Society B*, *371*(1701), 20150450. http://doi.org/10.1098/rstb.2015.0450

SERGI VALVERDE. Expert in complex systems with a PhD in Applied Physics and researcher at the Institute of Evolutionary Biology (UPF-CSIC), Barcelona (Spain), where he leads the Evolution of Technology Lab (ETL). His research group is a pioneer in the study of major evolutionary transitions by comparing biological and artificial systems. His multidisciplinary research integrates various areas of knowledge, from network theory to theoretical ecology and the computational simulation of evolutionary processes. He is a board member for the Catalan Network for the Study of Complex Systems (complexitat.cat). ✉ sergi.valverde@ibe.upf-csic.es