Some thoughts on the historicity of software
Liesbeth de Mol

To cite this version:
Liesbeth de Mol. Some thoughts on the historicity of software. 2016. hal-01357886

HAL Id: hal-01357886
https://hal.science/hal-01357886
Preprint submitted on 30 Aug 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Some thoughts on the historicity of software
Liesbeth De Mol, CNRS, UMR 8163 STL

In his paper, Nathan, argues for the significance of software maintenance within the wider history of software. There are certainly quantitative or economical arguments which favor this proposal. As Nathan has pointed out, "since the early 1960s to present, software maintenance costs have represented between 50% and 70% of all expenditures on software development". However, his main arguments are rather philosophical and rooted in a particular view on what software is: "software" -- far from being the result of an artistic act by someone who is free from any material constraints -- is "a tangible record, not only of the intentions of the original designer, but of the social, technological, and organization context in which it was developed". "Software is history, organization, and social relationships made tangible". In other words, software cannot be reduced to its code. Rather it is "an ever expanding category that grew not only in size and scale but also in scope". As such, developing software is "an unending and evolutionary process" which renders intelligible the fact that so-called "enhancements" or "adaptive maintenance" involve the majority of software maintenance. Perhaps most importantly, as a complex evolving heterogeneous system, embedded it has become impossible to start from scratch: software /is/ indeed history in the sense that whatever "software" we are using, it contains the traces of its own history, though these are mostly hidden from the user. Indeed, the Millenium bug, for instance, would never have been if one had not built layers on top of layers of software without critically assessing features of those earlier layers that affected the new ones, such as, in the case of the Millenium bug, the way dates are represented.

But if software has developed into a heterogeneous system, this does not mean that the historian should follow this view on software all the way through. Software understood as a heterogeneous system is in itself a historically developed notion. As Nathan indicates, it is rooted in the recognition in the 60s by "industry observers and computing service providers [of] the significance of [the] larger context". Before that time, software was "the collection of low-level tools used to construct and manage" what one would nowadays consider as software, the end-user application. This recognition of the larger context is the reason why, as Nathan remarks, Norbert Galler broadened the term software as follows: for the user "the total computing facility provided for his use, other than the hardware, is the software." However, this very interpretation of software is rooted in the user's perspective -- it is he/she who is

1 Ce texte a été présenté comme commentaire sur l'article de N. Ensmenger, When good software goes bad. The surprising durability of an ephemeral technology, présenté pendant la conférence Mice (Mistakes, Ignorance, Contingency and Errors in Science and Technology), à Munich, 2-4 Octobre 2014.
unable to perceive anything but software. It is the user who is unable to differentiate
between the elements of the software system.

But who exactly is this user? A neutral definition is that it is the person or
organization for who the "ware" is manufactured. But of course, since the user is
taken into account when designing software, the user cannot be neutral and is also
shaped by that very design process. A provocative comment on this "creation" of the
user is given by Dijkstra, a well-known computer scientist:

"[T]he computer user, as functioning in the development of computer products is not
a real person of flesh and blood but a literary figure, the creation of literature, rather
poor literature. […] [I]f you start to analyze the many character traits of that literary
figure, you discover that he is most uninspiring. He is stupid, education resistant if
not education proof, and he hates any form of intellectual demand made on him, he
cannot be delighted by something truly beautiful, because he lacks the education to
appreciate beauty. Large sections of computer science are paralyzed by accepting
this moron as their typical customer."

Just as software cannot be reduced to pure code, the user cannot be reduced to a
human social being or a managerial structure but rather should be seen as a
complex resulting from how people behave and use their computers, how the
industry wants them to behave and use their computers and how the actual "form" of
the technique asks them to behave and use that very same technique. Indeed, using
a GUI results in a very different kind of experience than using a text-based user
interface. In that sense, a historical study of the "user" is at least as much needed as
a study of the "ware" manufactured for his/or her use -- both go hand in hand.

"Software" and its crisis only came into existence with the development of higher-
level programming. This development was very much needed and arose from the so-
called "programming bottleneck": the problem that the time to set-up a problem on a
computer was completely out of sync with the speed of the machine, which, in its
turn was rooted in the fact that coding and hardware were inextricably intertwined --
coding was hardware manipulation, also from the "user's" perspective. There was
growing need for a more logical structure that not only allowed to program from a
more human-centered perspective but also to control the "automatic evolution of a
meaning", to put it in von Neumann's words, introduced by the speed of the machine.

This creation of programming languages as an in-between between computers and
humans implied the very creation of a user who cares less about the material
constraints that are nonetheless implied in any programming language and who /can/
take distance from the hardware by using a more symbolic way of communicating
with the machine. It also enabled a widening of the scope of high-speed computing resulting in a need for hiding the mathematico-logical nature of this very communication since also "non-technical" people could now use and thus had to communicate with the machine. Indeed, as the mathematical nature of computing steadily generalized from the scientific into corporate life, the construction of a user who does not need to know about the hardware and its language, became a fact. It is also that same user who contributes to the maintenance of software's historicity: "users" do not prefer to start from scratch or, to put it in McLuhan's words: "we tend always to attach ourselves to the objects, to the flavor of the most recent past. We look at the present through a rear-view mirror. We march backward into the future."

The failure of Bell labs project 9, intended as an improvement on the existing Unix system is one clear example of this kind of human behaviour. Even though it is generally acknowledged that it is an improvement it never replaced the Unix system. Eric Raymond, an advocate of the open source movement made the following comment on this failure in a paper titled "Plan 9: The way the future was":

"Plan 9 failed simply because it fell short of being a compelling enough improvement on Unix to displace its ancestor. Compared to Plan 9, Unix creaks and clanks and has obvious rust spots, but it gets the job done well enough to hold its position. There is a lesson here for ambitious system architects: the most dangerous enemy of a better solution is an existing codebase that is just good enough"

Even though software contains its own history and is thus constrained by it, this very historicity is mostly not critically re-evaluated when a new layer is added. This is not only rooted in the heterogeneous character of software or in the user's resistance to "start from scratch", but also in the very bureaucracy which supports this accumulation and does not wish to invest time and money in making the user, independent of whether he/she is a programmer or not, historically aware of what lies underneath the layers. To put it bluntly, it is not the purpose of Microsoft to develop a software which is historically transparent -- if that would be the case, the industry itself would not be sustainable.

From this perspective it is not surprising that it is not debugging but enhancement, defined as "responses to changes in the business environment" that is more important in maintenance practices: even though one bug can cost a lot, both economically as well as humanly speaking, adaptation to and of an ever less demanding user, at least intellectually speaking, is what keeps the business running. Maintenance as enhancement is not only an effect of the general Mahonean hardness of software as such but also of a business model. From that perspective, it
is not surprising that problems of error and, in general, of software malfunctioning, are more systematically studied in an academic context which is less guided by economics. One recent example is the development of a formally verified C compiler which was developed at the INRIA, a French research institution and intended, and I quote from the CompCert project website:

"[not] [f]or non-critical, "everyday" software [where] bugs introduced by the compiler are negligible compared to those already present in the source program [but for] safety-critical or mission-critical software, where human lives, critical infrastructures, or highly-sensitive information are at stake."

The history of software is indeed hard but exactly this feature requires an approach which is not only rooted in a notion of software that gives more attention to human agency, but also one that focuses on the more technical sides of software development. If we do not engage with software’s technicality, we assist in hiding its very historicity, which is "archived" in code. In my view, it is only by combining both approaches that we can make the very historicity of software which Nathan explained, more transparent up to the hardware level which /is/ intertwined with the software level, a fact which is painfully illustrated by the Millenium bug: the reason for using the two-digit format in the 60s and 70s was rooted in a hardware issue -- memory was too expensive, and saving a couple of bytes really meant something. It is also a fact that perhaps might become more obvious now that we are approaching the limits of miniaturization and hence of processing and storage power.

To end my thoughts on Nathan's talk, I would like to emphasize that this idea of rendering the historicity of software more transparent by engaging also with the technical details that underpin it is perhaps not so much a historical goal as it is a political goal: it might help us to make the "user" slightly less like Dijkstra's. In other words, as a historian of computing my motivation is not only to develop a decent piece of work for my fellow historians, but perhaps even more to develop a history of computing that contributes to a more critical reflection on the computational tools we all use in our everyday lives.