"'Tis but thy name that is my enemy". Some reflections on the art and science views on and in the history of programming.

Liesbeth de Mol

Liesbeth de Mol

“'Tis but thy name that is my enemy”. Some reflections on the art and science views on and in the history of programming.

The phrase “how x changed from an art into a science” is an often heard statement with respect to a broad range of x’s. While a study of the origins, contexts and histories of such phrases *in general* would be worthwhile in itself, I will instead focus on one specific such usage, that is, in the context of programming. More specifically, I will focus on artistic conceptions of programming in the work of two computer scientists which are, traditionally, understood as promoting a “scientific” view on programming. By doing so, I would like to show that the apparent semantical oppositions between art and science that are implied by such phrases, if considered true, run the risk of oversimplifying, if not hiding or misreading historical facts, by assuming a one-dimensional and unquestioned semantics of the words “art” and “science”.

So what do we assume and what do we want to achieve when characterizing a development in those terms? What kind of judgements do we make when we say that programming is an art or a science? And, perhaps most importantly, what kind of semantics do we actually assign to the words “art”, “science” and “programming” and what do we derive, by consequence, about their relations?

Since the basic principle behind our starting phrase is that there is a development from being an art to being a science, it is clear that art and science cannot be equated and that there is, apparently, a basic distinction if not opposition to be made. In order to see that, let us have a look at a rather well-known quote from John Backus:

> “programming in the early 1950s was a black *art*, a private arcane matter [...] General programming principles were largely nonexistent. Thus each problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and invention.”

Backus even talks about the “priesthood” of programmers “*guarding skills and mysteries far too complex for ordinary mortals*”. Here, the artistic semantically refers to craft techniques, and this is indeed also the meaning that is mostly used in some of the more historical works which have picked up this way of looking at developments in programming in the 1950s (Campbell-Kelly and Aspray 2004; Ensmenger 2010). It is then, supposedly, this “black art of programming” approach which is considered to be at the roots of the software crisis as well as of the attempts, both by academics and industrials, to control it, up to today, by developing a more “disciplined” field via the establishment of firm scientific, managerial and/or engineering foundations. The opposition between the two terms then becomes one where the former is bad and the latter is good practice.

At first sight, this echoes that other opposition that has ran in different shapes throughout our Western cultural history and which is that between Aristotle’s conception of épistémé and têchné; the old dichotomy between theory and practice. More specifically, the art versus science opposition implied here, seems to echo the medieval differentiation between the *artes mechanicae* or *artes vulgaris* for which “admission and training were

dominated by guilds or corporate bodies [and which were] oriented immediately towards the formation of practical occupational skills"[1] in contrast to the so-called *artes liberales* which were taught at the universities and were much more occupied with theoretical knowledge.

But of course, *art*, via its Greek reference to the word Têchné, does not just refer to craftsmanship, but also to "creative" works of art like paintings, poetry, literature or music. Even though the art-to-science conception of programming seems to refer *first* to art-as-craftmanship, this other meaning of "art", especially when contrasted with science, certainly was at play too, at least implicitly. From that perspective we can understand that Backus, seemingly negative about the "black art" was neither too enthousiastic about its "stuffy" and "disciplining" counterparts of science and management. Indeed, on the same page in which he described the black art in a negative manner, he also explained and complained that "*Today [1978] a programmer is often under great pressure from superiors who know just how and how long he should take to write a program; his work is no longer regarded as a mysterious art [...] Programming in the America of the 1950s had a vital frontier enthusiasm virtually untainted by either the scholarship or the stuffiness of academia. The programmer-inventors of the early 1950s were too impatient to hoard an idea until it could be fully developed and a paper written.*" In other words, being free, being creative, being inventive, etc were other semantic connotations that were typically associated with a programming-as-art conception. It is then science which received those other connotations of being cold, rational and formal. It is that understanding of science, in opposition to art, that one finds in Knuth's Turing award lecture titled *Computer programming as an art* in which he declared that "*the process of going from an art to a science means that we learn how to automate something.* [...]". But wasn't it Knuth who is put on the historical stage as one of the most important advocates of a computer science discipline called algorithmics? Isn't he one of those who wanted the field to be disciplined from an art into a mathematical science once it became clear that the programming field lacked and-needed disciplinary foundations?

In that same text, Knuth refers to programming as an art in the aesthetic sense: a program should be a piece of beauty and have "good style" ~~(which is left quite undefined)~~. A similar view can be found in one of Dijkstra's texts who, more than Knuth, insisted on the need for a mathematical foundation of programming~~.~~ Indeed, it is no accident that the idea of a software crisis that could only be resolved with more science is very much due to Dijkstra (Haigh 2010).

In one of his manuscripts titled *Some meditations on advanced programming"* Dijkstra ends the text with the following rather dramatic comment: "*The tool should be charming, it should be elegant, it should be worthy of our love.* […] *[T]he programmer does not differ from any other craftsman: unless he loves his tools it is highly improbable that he will ever create something of superior quality. At the same time these considerations tell us the greatest virtues a program can show: **E**legance and **B**eauty*." So, also here, we find an appeal to aesthetics but, also, to craftsmanship, a view which is in direct contrast with

1 Walter Ruegg, Themes, in: H. Symoens (ed.), Universities in the Middle Ages, p. 30.

earlier statements in the same text where Dijkstra regrets that "*programming has arisen not as a science but as a craft, as an occupation where man, under the pressure of the circumstances was guided more by opportunism than by sound principles".*

In another text, *A short introduction to the **art** of programming,* which was intended as lecture notes*,* Dijkstra relates the use of the phrase "the art of programming" to the teaching at a conservatory:

> "it is my purpose to transmit the importance of good taste and style in programming […] I feel akin to the teacher of composition at a conservatory: he does not teach his pupils how to compose a particular symphony, he must help his pupils to find their own style"

Here (quote used by Knuth) Dijkstra again refers to art in relation to the aesthetic quality of "good style" and sees this as the main task of teaching programming (and thus not, for instance, to teach a specific language). In another well-know text, the *Notes on structured programming* this aesthetic view is further specified and again explained metaphorically with a reference to music: *"elegance, clarity and the like have indeed marked quantitative aspects (as Mozart knew: many of his compositions that make one catch one's breath are misleadingly simple, they seem to be made just out of practically nothing!)."* Here, it becomes clear that Dijkstra's appeal to elegance and clarity are not just aesthetic but very much related to a basic problem of software: that is, its complexities. This is today still a major concern within software engineering practices and so developing methods to control it are quite basic. As he explains:

> "it is becoming most urgent to stop to consider programming primarily as the minimization of a cost/performance ratio. We should recognise that already now programming is much more an *intellectual* [m.i.] challenge: the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."

Thus, aesthetic properties like elegance are not there for beauty's sake but have quite a specific and practical purpose (see in this respect also (Daylight 2012)). This then is the relation between art-as-craftmanship and art-as-art: properties that are typically affiliated with aesthetic qualities; methods for "making" elegant programs are considered effective in creating "good" software. Moreover, Dijkstra uses this more "artistic" conception of programming against the rationalized practice of minimization of cost/performance ratios as one could find them both in managerial as well as in certain programming discourses. It is also and exactly at that point that Backus, as we saw, seems to re-appraise the artistic in the black-art phase of programming. Put differently, here we see how programming as an art *and* a science is used to go *against* the rationalism of program efficiency.

Knuth's case is quite different: not even mentioning his so-called bible of programming, the very fact that he decided to devote his Turing award lecture to programming-as-an-art is quite interesting. By the late 1970s Knuth's attention shifted to developing mathematical typesetting software since he was convinced that "*Mathematical books and journals do not look as beautiful as they used to*" (Knuth 1979). In his Gibbs lecture of 1979 for the AMS where he introduced the project, Knuth's previous references

to aesthetic qualities in relation to programming were turned into a practical problem of designing beautiful fonts: "*Of course it is necessary that the mathematically-defined letters be beautiful according to traditional notions of aesthetics. Given a sequence of points in the plane, what is the most pleasing curve that connects them?*" (Knuth 1979) Thus, the typesetting problem becomes Knuth's way of connecting the aesthetic with the technological, the art-as-art with art-as-craftsmanship and this is to be taken quite literal: as Knuth explains, the name of the system TeX derives from the Greek Têchné: "*Its emphasis is on art and technology, as in the underlying Greek word.*" (Knuth 1986).

By the mid-80s Knuth applies this viewpoint again but in quite a different manner: whereas the typesetting system TeX resulted from "*an application of computers to another field*" he would now apply the TeX system "*to the heart of computer science*" (Knuth 1984). That system was WEB. It combines programming with typesetting and develops a method of what Knuth baptized *literate programming.* The idea of the system was to attack the problem of documentation. Knuth believed that "better documentation of programs [...] can best [be] achieve[d] [...] by considering programs to be *works of literature."* (Knuth 1984). From that perspective, "[t]he practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style." The purpose then was to develop programs that are intended not just as a set of instructions for the computer but also as an explanation by humans to humans of what one wants the computer to do.

So what can we conclude from this short exploration of the work of two actors who are traditionally situated on the science side of the art-to-science conception of programming developments from the 1950s to now? While it is certainly uncontested that both Dijkstra and Knuth promoted a viewpoint on computing as being mathematical (each in different manners) and so contributed to the shaping of computing as a science, that need not mean that they necessarily opposed craftsmanship and artistry to science in their *actual* work. Perhaps Dijkstra might have done so rhetorically but even there it is clear that the semantics of "art" are much more complex then one might get from the usual "art-to-science" storyline. Indeed, they each considered art-as-têchné both as a means to achieve results that were aesthetically pleasing but also, because of that, better and superior to previous products. Thus, we see that the simple rhetorics behind the art-science opposition from which we started hides a much more complex history and semantics of which I have only scratched the surface here.

This brings me to a basic and open-ended question: is it because we have picked up a certain rhetorics from discourse and projected it back into our pasts, that we have reinforced it? While, as historians, we need to assume that our `storytelling' reflects historical reality, picking up the rhetorics from the object level without engaging in much detail with what lies underneath that rhetorical level, one runs a risk of misreading the *actual* historical practices and so, ultimately, to make the wrong assumptions about history. Indeed, isn't it exactly because we have so fathomed the 1950s as a period in which no real "science of programming" happens that we have been unable to see the historical continuities between work in the 1940s and 1950s and work in the 1960s? On a more general scale, one may well ask whether it is not exactly because many of us have decided

to inscribe history of computing within a history of technology rather than in a history of science that we glossed too easily over many of the more complex interactions between technology and science, one of the reasons maybe why we are still lacking a serious history of computer science?