Journal of Applied Logic 6 (2008) 545–552

Contents lists available at ScienceDirect

Journal of Applied Logic

www.elsevier.com/locate/jal

Dark programming and the case for the rationality of programs

Lars-Erik Janlert

University of Umeå, Department of Computing Science, Umeå, Sweden

ARTICLE INFO

Article history:
Available online 25 September 2008

Keywords:
Dark programming
Subjective method
Rationality of programs
Justifiability

ABSTRACT

Programming normally proceeds from subjective method to objective method: to program a task, *you* need to be able to do the task; at least "in principle." Some new techniques of producing programs, exemplified by evolutionary algorithm techniques, do not involve any such subjective method. Is this still programming, and are the results really programs? To answer, the concept of program is reexamined. It is argued that in addition to being causative and descriptive, a program must also be rationally justifiable, i.e., the specific structure of the program should be explainable by the (rational) ways in which it contributes to achieving the intended goal. Whereas traditional programming is rational in the sense that it provides the program with a rationale by its reliance on subjective method and problem solving, these new techniques of "dark programming" do not produce any rationale; moreover, the results are not restricted to be easily rationalized. Dark programs are not guaranteed to be proper programs, but dark programming can still be used as a tool in proper (rational) programming. The business of programming then takes a turn from problem solving in the sense of invention and engineering towards problem solving in the sense of explanation and research.

1. Traditional programming

At some point in teaching programming it is commonly claimed that programming is problem solving. But where exactly is the problem? Is the problem to compute square roots, or to create a program that computes square roots, or to define a procedure appropriate for the generation of a program that computes square roots?

In the practical application, when "the problem is solved," the distinctions will seem unimportant: once we have a program able to perform the task we can let the program be executed to have the task performed (or, if we have that procedure for generating a program able to perform the task, we can use the method to produce a program that can be executed to perform the task).

In the process of programming, the distinctions will also seem unimportant, because our usual approach to creating a program that can perform a given task is by way of some plan for performing the task that we could, in principle, carry out ourselves. If we want to make a program that computes square roots, we will need to invent some method that *we* could use to compute square-roots—and then go on to construct a program that works the same way, but without us. Such a method is *subjective* in the sense that a particular person grasps it and performs it, but the same method can certainly be shared among competent persons. By their very nature subjective methods necessarily make sense as a way to perform the task, to the person using it: not only do you know that it works, you also know what makes it work.

In short, the common view of programming as a problem-solving process involves going from a subjective method to a program, an objective method performed by a machine. *You* must be able to solve the problem for it to be solved by a program. Although it is somehow easy to forget [4], Turing's machine was designed to be an idealized model of a human

E-mail address: lej@cs.umu.se.

1570-8683/$ – see front matter
doi:10.1016/j.jal.2008.09.003

computer, and the informal notion of an effective method that is central in Church–Turing's thesis refers specifically to what a human being can do equipped with pen and paper, besides unlimited patience, perseverance, time, carefulness, etc.—corresponding to the important "in principle" qualification.

As a matter of fact, in real-world programming we are commonly dealing with subjective methods that we could not really use to solve the task *in practice*—only in our imagination, engaging in a kind of Gedanken experiment.[1] Practically useless but "in principle" working subjective methods are regularly turned into practically feasible objective methods in the everyday practice of programming.

Of course, programmers do not always have to create a suitable method from scratch. Often an existent program can solve a given task or subtask identified in the process of programming. The programmer's work will then largely consist in searching for and identifying a program capable of performing the requested task. If the process of identification goes sufficiently deep, it will amount to a reconstruction of the underlying subjective method on the part of the programmer looking for a solution. But the goal in reusing software is obviously to avoid having to go that deep: clear descriptions of what the program achieves, the conditions for its proper use, etc., should be sufficient for quick identification and correct application; you should not have to worry about exactly *how* it does what it does.

Still, subjective problem solving and methods are involved in creating those software building blocks and ready-made programs, to begin with. The situation is analogous to the division of linguistic "labor" theorized by Putnam [16]: I can (responsibly) use the word "beech" without knowing an exact definition or being very confident in recognizing a beech tree; but somewhere there are specialists who can do precisely that. Similarly, in programming there is a social structure that makes it workable and acceptable not knowing the details and rationales of the methods, instead relying on the expertise and testimony of others as to their effects and conditions for use.

In short, traditional programming is as a matter of fact founded on a basis of subjective methods.

2. Dark programming

There are now other ways of producing programs, however, that do not seem to require that *anyone* has had a clear comprehension of how the problem is solved, in the sense of having a subjective method. Two examples of such techniques that will be discussed here are: training an artificial neural network to do the task; using an evolutionary algorithm to evolve a program.

An artificial neural network (ANN) consists of a number of simple computational units connected with each other in a network [14]. Each unit computes an output signal transmitted through its outgoing connections, which is some very simple function (e.g. the sum) of the signal values on the unit's ingoing connections. The connections have different weights regulating the signal transmission from one unit to another. Certain units are designated as input units; certain units are output units. The network as a whole can be considered to compute a function from the vector of input unit values to the vector of output unit values; which function depends primarily on the weights of the different connections. E.g. if the input units represent pixels in a photo-sensitive array, the task may be character recognition, i.e., to determine what hand-written letter or word is scanned, and the result can be represented as a certain pattern of output unit values (see Fig. 1). Typically the network is *trained* to perform the task, by repeatedly and automatically evaluating its performance and adjusting the connection weights to gradually improve the performance, using some particular learning algorithm. While a fully trained ANN usually would not be *called* a "program," it clearly may perform the same task as a regular program.

Evolutionary algorithm techniques imitate or are inspired by biological evolution. There is great variation in the degree of similarity with, and the presence of known dissimilarities from, natural evolution, but a common pattern is the generate–select cycle of Darwinian evolution. A population of entities are culled by selection pressure, the remaining entities are the basis for regeneration of the population, usually involving random operators, crossover and mutation on genetic material associated with each entity. In particular, evolutionary techniques have been applied with *programs* as entities, producing programs that are able to perform a predefined task, e.g. sorting sequences of numbers [11,12] (see Fig. 2).

Similar techniques that bypass the employment of subjective methods can be found e.g. in machine learning, in artificial intelligence planning [19], and in swarm intelligence [2]. Once you have a program resulting from some such scheme, and assuming it works and indeed reliably performs the task, you certainly have a solution in the sense that you can have the program executed to obtain the sought-for result. But it is quite possible that you are unable to comprehend the program and unable to perform the method it implements in any other way than mechanically simulating the execution by hand, step by step. That is, a subjective method is lacking.

If not exactly a black box, the end product in these cases will at least initially be *dark* in the sense that there will be no prior understanding of it, so we might perhaps call these alternative approaches to produce programs "dark programming." That a program is "dark" in this sense is entirely compatible with a higher-level guarantee that it works; there may well be some sort of convergence proof, so that we know for certain that the program indeed performs the task as it should. In other cases of dark programming, there may be no such hard proofs available, but perhaps practical proofs or statistical proofs.

[1] Rotman [18] presents a model for mathematical reasoning, which is of interest for understanding how the gap between subjective and objective method is bridged.

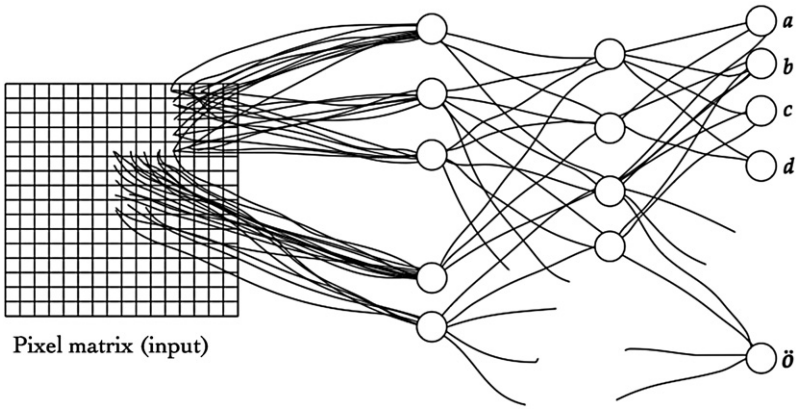

Fig. 1. Figure 1 Sketch of an artificial neural network that recognizes different characters.

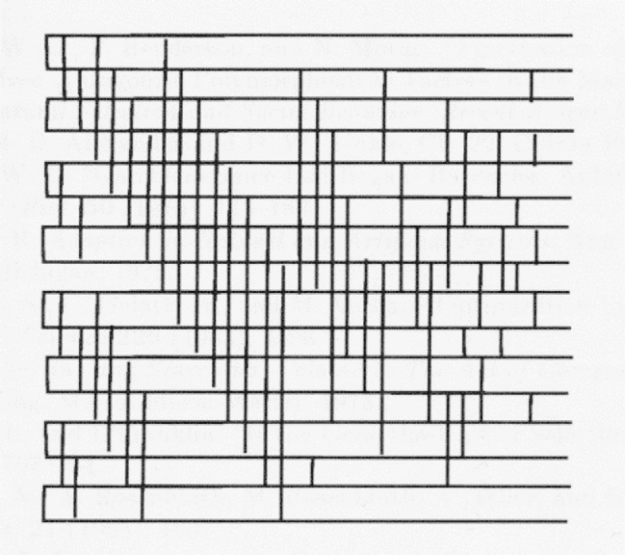

Fig. 2. This diagram from Hillis [11] represents a program sorting 16 numbers, produced by evolutionary techniques. Each number is represented by a horizontal line. The unsorted input is on the left, the sorted output on the right. Each vertical line represents the operation of comparing and exchanging two numbers (at the endpoints) if they are in the wrong order.

3. Sources of incomprehensibility

One cause of lack of comprehensibility may be that the "program," the generated entity that solves the task, simply is not an expression in some programming language. ANNs are an example: a trained ANN is not a program if you require of a program that it not only is (can be) the cause of a certain, intended computational process but that it also *describes* this process in an implementation-independent (i.e. causal-relation independent) way. Given the requirement of descriptiveness, the only way to understand an ANN as a program would be to find a program in some programming language, which this ANN can be interpreted to be the implementation of.

When the "program" formally is an expression in some programming language (which would typically be the case when evolutionary techniques are used to produce it), at a deeper level it may still be incomprehensible for the human reader. Any programmer would agree that to *understand* a program, requires much more than just being able to *execute* it, to simulate its behavior by hand. Observing and reflecting on the results of such simulations can certainly help to improve understanding, but the execution itself does not call for prior understanding. The *depth of intention*[2] required to perform as a processor is arguably nil—all that is needed for an ordinary computer is the capability to correctly identify and react to the tokens—whereas the depth of intention required to understand the program should at least *approach* the depth of intention of the programmer that created the program in the first place.

[2] Using the term of Arne Naess [15].

We know how easy it is to obfuscate the reading of a program simply by changing or "anonymizing" the identifiers; some of the intentions and understandings of the original programmer is normally conveyed to other programmers via identifiers, by choosing informative names, and also via various (programming) cultural conventions and idioms, not to speak of explicit comments and separate, associated documentation. This goes to show that programmers have realized that it is sometimes important for other programmers to understand the subjective method they have used; they have found that the completely unadorned program "as perceived by the computer" is inadequate to convey this understanding; and they have thus developed various complementary devices and channels for communication [1]. Also in the privileged position of the creator of the program, these devices serve as important reminders during the construction process and after: our memory is short, our span of attention limited, so external representations are helpful.

Still, what you have created yourself you are supposedly in a special position to understand (it is after all your own subjective method that is being transformed into an objective construction as you build the program).[3] This is a popular idea that readily extends into a general and widely used strategy of seeking understanding and explanation of objects and phenomena, whether in nature or in culture: trying to understand them *as if* they were something we could imagine having created ourselves for some imagined purpose.

To conclude, if already what some other programmer has produced may cause considerable trouble to understand, we should not be very surprised if trying to make sense of what no human being has devised may turn into a veritable research project.[4]

4. What qualifies as a program?

In order to decide whether dark programming qualifies as programming and the results can be considered to be programs, we need to examine and reflect on the requirements we should have on a program. What *is* a program?

I propose that the following three criteria for something being a program should be taken as necessary:

1) *causative*—the program causes the process;
2) *descriptive*—the program describes the process;
3) *justifiable*—the program has a rational explanation in terms of the goal of the process; i.e. the structure of the program can be explained by the (rational) ways in which it contributes to achieving the goal.

These criteria are formulated in terms of the program's relation to the process programmed, the intended process serving to satisfy certain, intended goals. I expect criteria 1 and 2 to correspond well to generally held and recognized beliefs [8,22], whereas criterion 3 may have some news value.

The requirement of causation (1) is necessary to distinguish programs from the more general notion of a *plan*. Plans have been used to design and guide future behavior for as long back as there are historical records, and probably long before the invention of writing; many would take the ability to make and use plans as part of what it is to be human. A general problem with plans, especially plans made up by someone else, is that people tend not to follow them. They are careless, they misinterpret, they forget, they cheat, they tire, get disinterested and may decide to do something else. Programming solves this problem by making the plan directly cause the intended process without human intervention, thus eliminating the gap, the troublesome leeway, between plan and execution. Viewed from this perspective, the program is the consummation of industrial society.

The requirement of description (2) is necessary to distinguish programs from the general notion of a *cause* of some specific behavior. The spring of a mechanical toy is a cause of the movements of the toy, but it does not describe them. In Herbert Simon's ant parable [21], the topology of the beach is the cause of the ant's irregular, complex path, but it does not describe it. Description minimally requires a symbol system: a symbol scheme and a field of reference [10]. There is little doubt that the practice of programming as we know it, would cease to exist if programmers did not have programming languages allowing them to take a shortcut and deal with the intended process directly through language constructs, and instead were compelled to think in terms of the causal chains set off by the material program.

The requirement of justification (3) mirrors the assumption that a subjective method is *not* just a recipe: it also needs a rationale; it must be a reasoned method, which explains why manually simulating the execution of a functional "program" does not in itself qualify as a subjective method. I believe the requirement of justification is in fact tacitly embraced by most programmers; the reason it has not been spelled out more clearly may be that until recently it has been taken for granted that programs only arise through a conscious, purposeful process. Whenever we care to design something, per definition there is purpose, and surely we expect there to be motives and reasons for the decisions made in the designing process, even when the designer is unwilling or unable to give an exact account of them. The fundamental fact that purposes and rationales can play no role in executing a program, may possibly foster the notion that purposes and rationales are not part and parcel of the program, but belong rather to a somewhat mysterious, creative process that should be left out of the

[3] Compare Vico's famous dictum *verum ipsum factum*—to really know a thing, you need to know how it was made.

[4] To be sure, it may happen that dark programming results in a program that is identical or very similar to a program that already exists in the (traditional) programming literature; that should pave the way for a simplified sense-making process.

picture when assessing a program qua program; a notion that also might find support in the (debatable) analogy with the demarcation between context of discovery and context of justification in philosophy of science.[5]

Although the practice of traditional programming, its institutional character and its ultimate reliance on the understandings of human beings might lead one to think otherwise, I want to emphasize that the justification criterion does not require that some intentional being currently understands the rationality of the program; just that the program is amenable to a rational explanation. I consider the property of being a program to be a brute fact rather than an institutional fact, in the terms of Searle [20].

What, one may ask, is then the relation of the justification requirement to the formal verification movement's call for mathematical proofs of programs?[6] The goal of the formal verification movement was (and still is) to prove the proper function of the program. It is a different goal than what is at stake here: the rationality of the program. A proof that a program works is not a proof that a justifying rationale exists (and a proof of the existence of a rationale is not a rationale itself). Nevertheless, different techniques that have been used in formal verification presuppose the employment of rationales; in that respect the situation is not very different from the reliance of traditional programming on subjective methods, verificationism is rather an amplification, a hardcore version of traditional programming.

Above, I have argued that being rationally explainable is part of what being a program should *mean*. A second argument for the justification requirement is that it puts constraints on the program that serve to give it properties valuable from an engineering point of view.

Taking Herbert Simon as an authority on the subject, engineering is characterized by modularity, separation of functions, hierarchies and nearly decomposable systems [21, pp. 183–208]. These constraints seem to be more or less necessary for us to be able to create intentionally—"by design"; they are certainly deeply involved in the practical application of our current notion of rationality as goal-directed, calculated choice. The subjective method developed in traditional programming has to satisfy the rationality requirement, and *to manage* that, the programmer is compelled to abide by the good engineering constraints. Importantly, "good engineering" generally implies a certain necessary sacrifice of compactness and efficiency in the end product for the sake of attaining rational perspicuity. We know, e.g., that the most compact program is necessarily also the maximally unstructured and opaque [13].

Since dark programming by definition does not depend on or involve any subjective method and use of rationales, it is not constrained to obey sound engineering principles.

There are numerous desirable properties—predictability, testability, modifiability, amenability to location and correction of errors and performance deficiencies, to verification and validation efforts and attempts to assess how good the design is, enabling division of work, reuse of solutions in future designs, etc.—that critically depend on modularity, separation of functions, hierarchies and near decomposability. All of these properties seem directly related to the process of developing the design and with the prospects of continued maintenance and future modifications and development of new designs. The good engineering constraints appear to be constraints by which designs, because of cognitive limitations, or maybe even epistemological, in fact *are*, and perhaps *have to be*, developed by human designers (and possibly rational agents in general).

Since dark programming neither depends on the assistance of rational programmers in the process of producing its result, nor is targeted for future modification, development, or reuse of parts, it is not subject to restrictions that have to do with the limitations of human programmers and program development in a longer perspective.

In conclusion, whereas traditional programming (ideally) satisfies the justification criterion thanks to its reliance on subjective methods, dark programming fails to provide justifications. Dark programs are not necessarily programs.

5. Rationalization and its limits

Dark programming does not deliver the kind of explanation requested by the justification requirement at the time a "program" is produced—but it may conceivably be supplied after the fact. Only then would we be able to confer the hitherto dark program the status of a proper program, according to the proposed criteria. The "darkness" of a program, if we want to use such a term, thus relates to our own state of knowledge, our present uncertainty as to the satisfaction of the criteria. A dark program that has resulted from dark programming either is a proper program, we just haven't satisfied ourselves that it can be rationally explained yet—or it is not a program, because it cannot be rationally explained in the manner required by the justification criterion.

To produce a post factum justification of a dark program is like making sense of a design with a known purpose: we know the general cause, we know the end result (the intended goal)—what we need to do is distinguish the important features of the cause, and get to understand which rational design considerations motivated them. We want to be able to view the specifics of the cause as a rational answer to the question of how to produce the desired result, given the circumstances.

This brings us to *reverse engineering*—the process of discovering the technological principles of an artifact through analysis of its structure, function and operation. Reverse engineering is a rather popular term open to different interpretations [3],

[5] Reichenbach [17]. As it happens, in this analogy, the justification criterion ends up matching the context of discovery, and the other criteria match the context of justification!

[6] Dijkstra [7].

but I am thinking specifically of a definition along the line of Dennett's [5]: "the interpretation of an already existing artifact by an analysis of the design considerations that must have governed its creation." To this I would like to add as a corollary tying in with the earlier discussion, that reverse engineering is to form an understanding of something which mimics the understanding you would have had if you had designed it yourself; reverse engineering can be viewed as an act of make-believe engineering. It means that you not only understand how it works and can make (limited) predictions of its behavior—it also means that you understand the design decisions, the rationality of *why* it is designed the way it is (just as in the justification criterion). Arguably, reverse engineering must abide by the same constraints as ordinary, i.e. forward, engineering to be acceptable as make-believe forward engineering.

We cannot take for granted, however, that it will be easy, or even always possible, to rationalize a "program" created by dark programming. There is an important difference from reverse engineering in its original, non-metaphorical, meaning: the assumption that what you are trying to analyze *is* the result of a process of rational design.

Let us pick the evolutionary approach as example to see the challenge it presents for reverse engineering (and I believe that other varieties of dark programming are beset with similar problems). How much is the work of evolution (natural or artificial) like the work of a designer? We know that natural evolution has no foresight. There is no engineer and no rational mind behind naturally evolved "designs." It is clear that natural evolution still manages to promote the formation of hierarchies and nearly decomposable system: we find that biological systems often can be partly and approximately understood as if they were properly engineered. But there are notable exceptions even if we exclude from consideration cases such as remnants, "scaffoldings," from the development process, and the occasional completely non-functional component or feature, although they may certainly be confusing, too [6].[7] Nature does not have to respect good engineering principles, because it does not involve the assistance of intentional beings, and in fact biological bricolage serendipitously exploiting odd "side effects," producing non-modular solutions, multiple functions and functionality crossovers, etc., seem to be involved in important parts of actual natural designs.

One example at the very core of biological designs is the case of overlapping genes [23]. In genetic material (DNA or RNA) consecutive triplets of nucleotides codify a sequence of amino acids (a protein), each triplet standing for a single amino acid. This gives three different options for dividing the nucleotides into triplets, three different "reading frames," which opens for the possibility that two different genes may share the very same genetic material, by being coded using different reading frames. They would overlap physically, but inhabit different phase shifts, so to speak. Indeed, overlapping genes do exist, and far from being an unusual curiosity, the human genome contains thousands of them. It goes without saying that trying to modify one gene without disrupting the other would be like changing a word in a crossword puzzle without corrupting the other words sharing the same letters. Such cases of "bad engineering" are the more inscrutable. I suppose it is in principle possible that there even can be natural designs (that we can understand as serving certain functions) that we will never manage to entirely make sense of as rational designs.

In the particular case of using artificial evolution as our device for dark programming, we might try to take countermeasures by adding selection pressure for comprehensibility (or more specifically, various good engineering properties, as above). It could work, but would it not lay too heavy a hand on creativity? Natural evolution does not have to worry about being "creative" enough; since it is free from predetermined purpose, there is no way it can fail. This is however a distinct possibility for artificial evolution with a set goal. As a general principle, it would probably not be a good idea to select against everything that an engineer would not readily recognize as rational. Real progress does not necessarily occur while sticking strictly to sound engineering principles—the road to an innovative, and eventually rational, solution may not go via a series of incremental intermediary steps each of which can be rationally justified before we have arrived at a workable solution.

There is the romantic idea of the creative leap, supposedly followed by the often arduous work to rationalize, and even make it appear as if rationally engineered to begin with.[8] Without subscribing to that idea, we can view a dark program as a given that we attempt to understand as something made; from that point of view it represents a "creative leap" that needs to be backed up by a proper rationale. Developing programs the traditional way, checks for the rationality of the designs under construction are frequent, usually only call for minor revisions and thus may not draw much attention. Dark programming, in contrast, openly challenge our rationalizing ability in presenting us with a considerable chunk of new code that does not come with a history of past, rationally supported partial results.

I conclude that illuminating a dark program will usually take some work, the task may be arbitrarily hard, and success is not guaranteed.

6. "Gray" programs and the future of dark programming

Traditional programming and dark programming are two—rather different—practices of programming; the first well established, the second more a possibility than a fact. Two general questions come to mind: How different from traditional programming is dark programming? Given that dark programming does not automatically produce programs, can we responsibly use the products without bothering to establish their rationality?

[7] Compare the use of junk code to thwart reverse-engineering attempts by software competitors.

[8] "Meine Resultate habe ich längst, ich weiss nur noch nicht wie ich zu ihnen gelangen soll"—I have had my results since a long time, I just don't know yet how I will arrive at them—Gauss, as quoted by Friedell [9, p. 394].

The second question has a straightforward and simple answer: yes, we can, in so far as the "program" has been proven to work to our satisfaction (practically, statistically, by some convergence theorem, etc.). We can of course still see to it that the generating method is a program in the full sense: rationality would be guaranteed at the metalevel, and we could take comfort in the thought that these things at least are the *outcome* of rationally designed processes. From the practical point of view of immediate use of the end product the situation does not seem to be very different from traditional programming. The main difference is in its usefulness in the longer run: a dark program does not have reusable parts, does not allow modification and development.

Regarding the first question: one may perhaps suspect that it will be difficult to draw a clear line between traditional and dark programming. One could claim that there is an extended gray area between proper programs as defined here, and the products of dark programming. A program may by and large be recognized to be a proper program but yet contain some part that remains dark; i.e., it would be certified to be a proper program but for this not (yet) rationalized part.

As a possible example of darkness in traditional programming, consider the typical relation between source code and compiled (and optimized) code: the compiled code is globally incomprehensible, more or less, but we accept it by referring to the fact that it is the product of comprehensible, justified programs: the source code plus the compiler program. But the similarity seems to stop there: in this case we have two programs with same purpose and similar design. Small changes in the source code are systematically reflected in the compiled code; source code and compiled code are reasonably homomorphic. In dark programming, by contrast, the generating program and the product do not have same purpose or similar designs; they are not at the same level even. And yet, if we do not set any limits for what kinds of optimization techniques are allowed we must in principle be prepared to face (initially) inscrutable, incomprehensible compiled code, as a result of using, e.g., evolutionary methods in the optimization process. The process would then undeniably shift towards dark programming—but only with respect to the compiled code: it would not touch the rationality of the source code, not affect its good engineering properties and amenability to modification and development.[9]

As a second example, there is the not uncommon case of traditional programming gone out of hand in a complex mess that eventually no one really understands any more; the original rationales for various parts of the program apparently having succumbed to the pressure on the greater whole to deliver and keep on delivering.[10] In large programming projects it seems almost as if forces are at work to "darken" what started out as certified programs. The problems with programs "gone wild" are similar to those of dark programming, but with a difference: there will have been rationales to begin with, rationales now partly hidden and corrupted, but hopefully possible to restore and bring back in modified form. Again, I should emphasize that simply forgetting the rationale does not turn a proper program into a non-program, but it may lead to uncertainty about its rationality, and thus make it dark; moreover, the lack of access to a justification, is likely to introduce or increase uncertainty with respect to further developments and modifications.

In conclusion, there is potentially a spectrum of programs that vary in the degree to which they are known to meet the justification criterion. There is little doubt, however, that traditional programming embraces justification as an ideal, actively uses rationales in program development, and is continually fighting to stay justified, although sometimes and temporarily losing minor battles. Wherever programs are poorly justified, there will be good reason to push towards fuller justification, regardless of how the present lack of rationale can be explained. Dark programming, in contrast, does not bring justification with it; it will have to come after the fact, and very deliberately so.

Even though it may appear tempting to use methods that promise useful results while eschewing the hard labor of traditional programming, wide-spread use of dark programming techniques as a device for producing ready-to-use programs would meet resistance given my assumption that practicing programmers implicitly embrace the justification criterion. As an added tool in the programming process the prospects for dark programming are brighter: dark programming can be used to project possible solutions, followed by reverse engineering and reengineering to justify and consolidate results and partial results. Used in this manner dark programming will tend to shift the business of programming from problem solving in the sense of invention and engineering towards problem solving in the sense of explanation and research.

Finally, there is the scientific point of view. If something works but we do not understand how, that is certainly an incentive to find out. Science wants to know, and we have learned that in the end it often also will prove useful to know. Dark programs represent a scientific challenge and there is reason to believe that finding the rationales of dark programs (or their more easily rationalized approximations) will further not only computer science but also the practice of programming. Indeed, as our artifacts become more complex, powerful—and harder to understand—it becomes more important to understand them in order to be able to continue progress.

7. Conclusion

I have argued that being rationally justifiable is an important part of what it is to be a program. In traditional programming, as in engineering in general, rationality is a requirement for intentional design. The rationality requirement in

[9] Then again, if some "dark optimization" should turn out to produce some surprisingly efficient results, we would certainly be interested in how that happened, and want to know in more detail how the compiled code works.

[10] Let me point out that it is not really required that any single person understands the whole: what is required is just that it all has a rational explanation. The problem of large projects is rather to establish the existence of a rationale for the total program by coordinating a number of different partial rationales for different parts of the program, and while those parts are undergoing continuous change.

developing a design leaves certain telling traces in the result, such as modularity and separation of functions. The new, non-traditional techniques of producing programs that I have called dark programming do not provide rationales and thus are not guaranteed to produce programs, strictly speaking. Dark programs, not being produced under the constraints of rationality, do not necessarily have the features typical of good engineering, and may not succumb easily to attempts at reverse engineering, since they were not engineered to begin with. Still, dark programming can be seen as a new tool for programming in combination with reverse engineering and reengineering to rationalize the end result.

Acknowledgements

Thanks to two anonymous reviewers for their much appreciated and useful comments and criticism!

References

[1] H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs, second ed., The MIT Press, Cambridge, MA, 1996.
[2] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Oxford University Press, New York, 1999.
[3] E.J. Chikofsky, J.H. Cross, Reverse engineering and design recovery: A taxonomy, IEEE Software 7 (1990) 13–17.
[4] B.J. Copeland, Computation, in: L. Floridi (Ed.), Philosophy of Computing and Information, Blackwell, Oxford, 2004.
[5] D.C. Dennett, Cognitive science as reverse engineering: Several meanings of "top-down and "bottom-up", in: D. Prawitz, B. Skyrms, D. Westerståhl (Eds.), Logic, Methodology and Philosophy of Science IX, North-Holland, Amsterdam, 1994, pp. 679–689.
[6] D.C. Dennett, Darwin's Dangerous Idea, Simon & Schuster, New York, 1995.
[7] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[8] J.H. Fetzer, Program verification: The very idea, CACM 31 (9) (1988) 1048–1063.
[9] E. Friedell, Kulturgeschichte der Neuzeit, C.H. Beck, München, 1931.
[10] N. Goodman, Languages of Art. An Approach to a Theory of Symbols, The Bobbs-Merrill Company Inc, Indianapolis, 1968.
[11] W.D. Hillis, Co-evolving parasites improve simulated evolution as an optimization procedure, in: C.G. Langton, C. Taylor, J.D. Farmer, S. Rasmussen (Eds.), Artificial Life II, Addison-Wesley, New York, 1990.
[12] J.R. Koza, F.H. Bennett, M.A. Andre, D. Keane, Genetic Programming III: Darwinian Invention and Problem Solving, Morgan Kaufmann, San Francisco, CA, 1999.
[13] M. Li, P. Vitányi, An Introduction to Kolmogorov Complexity and Its Applications, Springer, New York, 1997.
[14] J.L. McClelland, D.E. Rumelhart, Parallel Distributed Processing, vols. 1–2, The MIT Press, Cambridge, MA, 1986.
[15] A. Naess, Interpretation and Preciseness: A Contribution to the Theory of Communication, Det Norske Videnskapsakademi i Oslo, Oslo, 1953.
[16] H. Putnam, The meaning of 'meaning', in: H. Putnam (Ed.), Mind, Language and Reality. Philosophical Papers, vol. 2, Cambridge University Press, Cambridge, 1975.
[17] H. Reichenbach, Experience and Prediction, University of Chicago Press, Chicago, 1938.
[18] B. Rotman, Ad Infinitum. The Ghost in Turing's Machine, Stanford University Press, Stanford, 1993.
[19] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, second ed., Prentice Hall, Upper Saddle River, NJ, 1998.
[20] J.R. Searle, The Construction of Social Reality, Simon & Schuster, New York, 1995.
[21] H. Simon, The Sciences of the Artificial, third ed., The MIT Press, Cambridge, MA, 1998. (First edition 1969.)
[22] B.C. Smith, On the Origin of Objects, The MIT Press, Cambridge, MA, 1996.
[23] V. Veeramachaneni, W. Makalowski, M. Galdzicki, R. Sood, I. Makalowska, Mammalian overlapping genes: The comparative perspective, Genome Research 14 (2004) 280–286.