The epistemology of programming language paradigms

First Author
Department
University
first.author@university.where

First Author
Department
University
second.author@university.where

May 9, 2013

Abstract

After the structured program theorem by Böhm & Jacopini (1966), different computer programming languages spread out. In fact, in spite of the fact that most programming languages are Turing-complete and hence the same expressive power, some programming languages are more equal than others in accomplishing specific goals, although no particular programming language is better than the others in absolute terms. This paper addresses the research question of what changes in terms of the programmers' knowledge in choosing one programming language instead of another. The method of levels of abstraction by Floridi (2008) is used in analysing four prototypical paradigms of computer programming: procedural, functional, object-oriented, and logic-based. The analysis clarifies the epistemological commitments adopted by the programmer within each paradigm, and in particular the levels of abstraction that get hidden. Some consequences in terms of programmer's responsibility (ethics) and in terms of beauty of source code (aesthetics) are envisaged in the conclusion.

Extended abstract

The history of modern computer programming languages can be traced back to the structured program theorem by Böhm & Jacopini (1966) and the *j'accuse* against the GOTO statement by Dijkstra (1968). In October of the same year, the conference organised by the NATO Science Committee introduced the concept of 'software engineering', and an IFIP Working Group on 'Programming Methodology' was established. As recalled by Dijkstra (2001): '[IBM] did not like the popularity of my text; it stole the term "Structured Programming" and under its auspices Harlan D. Mills trivialized the original concept to the abolishment of the goto statement.' It became evident that computer programming ought to be more solid, both theoretically and practically, also because software projects were becoming more complex, involving an increasing number of programmers, as for instance the IBM System/360 family and in particular the OS/360 (Brooks 1995).

One of the strategies adopted by computer scientists to cope with this growing complexity was to design new programming languages, at different Levels of Abstraction (LoAs). The method of LoAs—fully explained and defended in Floridi (2008)—can be used to describe programming in terms of informational organisms (inforgs): the programmer is the informational agent, while the computing machinery is the artificial artifact. A software project is an *infosphere*, including processes and mutual relations among the inforgs directed to the same goal. Under this perspective, the source code, i.e., the *observable*, is the main LoA acting with the Levels of Organisations (LoOs) of the machinery, i.e., the hierarchical structure of hardware *de re*. Therefore, the choice of the programming language is crucial, as it determines the epistemological approach sustaining the programmers' goals, which is identified as a Level of Explanation (LoE) by Floridi (2008). Unlike LoAs and LoOs, LoEs do not really pertain to the system, rather they are an epistemological lens through which the informational agent(s) approaches the goal of programming.

Examining computational inforgs—where the artificial artifact is a Von Neumann Machine (VNM)—Gobbo & Benini (2013) propose to look at the history of modern computing in terms of *information hiding*. Here, the scope is narrower, the choice of the programming language in terms of LoEs being the research question; however, the concept of information hiding can be usefully applied straightforwardly. In the early days, there was only a machine-tailored assembler letting programmers write one-to-one machine-readable instructions. Afterwards, a fundamental LoA was introduced by Backus during the design of FORTRAN (Backus 1978) and its implementation via a compiler, i.e., the computer program that translates the source code into machine code. In fact, he provided a formal notation that became the standard to describe programming languages: the Backus-Naur Form (BNF) abstracts over the language, allowing to compute on its structures, and thus it is a new LoA in computational inforgs. The next LoA in programming has been introduced after Böhm & Jacopini (1966), a result that permitted to hide the way the machine interprets the flow of control, and to change it to something which can be easily analysed mathematically. This result opened the door to the construction of a plethora of programming languages, each one adding LoAs to hide or change the behaviour of some aspect of the machine.

In fact, to cope with the growing complexity of the problems throughout the history of computing (Ceruzzi 2003), computer scientists (on the theoretical side, e.g., McCarthy's LISP) and informaticians (on the practical side, e.g., COBOL) construed languages in order to facilitate the modelling the possible solutions of a given family of problems. We can classify programming languages in few major paradigms, according to the information got hidden. Paraphrasing the three layered description of programming by Hofstadter (1979), we can consider the source code as the novel, the programmer being the novelist, while the programming language is the literary genre of the novel.

If the VNM should be step-by-step programmed, procedural languages such as C or Pascal, the direct descendants of structural programming, are apt to the goal: as underlined by White (2004) in Floridi (2004): 'most programming languages allow programs to perform actions that change the values of variables, or which have other irreversible effects (input or output, for example); we say that these actions have side-effects'.
On the contrary, if the problem is better conceived as a formal entity, the classical paradigm of mathematics can be used. In functional programming, the modelled world is described in terms of pure functions, taking as the LoE the tradition of computation as application of mathematical operations.
Another approach is by the introduction of the concept of *object*, which is conceived originally on a different LoE, philosophically based on Leibniz's monads and the notions of 20th century physic and biology (Kay 1993). It's a more sophisticated world, as the problem is split into different (virtual) VNMs that communicate one to the other by messaging and changing the local state of the object.
Finally, the fourth approach completely hides the VNM under a logical theory, so to let the program forget the algorithmic details—covered by the artificial reasoner—and modelling the problem in logical terms. Prototypically, this is the strategy followed by the PROLOG language, based on the Horn clauses and unification. In practice, the procedural side of programming cannot be eliminated completely (Lloyd 1984).

In this paper, we propose a taxonomy of these four paradigms (procedural, functional, object-oriented, logic-based) in terms of LoAs based on information hiding. This taxonomy provides the structure on which each language' constructions can be canonically simulated by another language.

In conclusion, it is evident that each programming language envisages a 'vision of the world' which is suitable for some classes of problems. When dealing with really complex problems, which happens in most contemporary software design, rarely a single language has the right features to model the whole problem. In this paper, we wanted to contribute to clarify the epistemiological statements behind the major classes of programming languages, together with their mutual relations. By providing a taxonomy, it becomes possible to implement notions from one language into another, simulating the LoAs and the features not originally present.

References

Backus, J. (1978), 'The history of FORTRAN I, II, and III', *SIGPLAN Not.* **13**(8), 165–180.

Böhm, C. & Jacopini, G. (1966), 'Flow diagrams, turing machines and languages with only two formation rules', *Communications of the ACM* **9**(5), 366–371.

Brooks, Jr., F. P. (1995), *The mythical man-month (anniversary ed.)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ceruzzi, P. (2003), *A history of modern computing*, History of computing, MIT Press.
URL: *http://books.google.com/books?id=x1YESXanrgQC*

Dijkstra, E. W. (1968), 'Letters to the editor: go to statement considered harmful', *Commun. ACM* **11**(3), 147–148.

Dijkstra, E. W. (2001), What led to "notes on structured programming". circulated privately.
URL: *http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1308.PDF*

Floridi, L. (2008), 'The method of levels of abstraction', *Minds Mach.* **18**, 303–329.

Floridi, L., ed. (2004), *The Blackwell guide to the philosophy of computing and information*, Blackwell, London.

Gobbo, F. & Benini, M. (2013), 'From ancient to modern computing: A history of information hiding', *IEEE Annals of the History of Computing* **99**(PrePrints).

Hofstadter, D. R. (1979), *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York.

Kay, A. C. (1993), 'The early history of smalltalk', *SIGPLAN Not.* **28**(3), 69–95.

Lloyd, J. W. (1984), *Foundations of logic programming*, Springer-Verlag New York, Inc., New York, NY, USA.

White, G. (2004), *The Philosophy of Computer Languages*, in Floridi (2004), chapter 18.