California Polytechnic State University, Department of Computer Science,
Technical Report Number CPSLO-CSC-02-01, October 28, 2002.

The Death of the Programmer?

Phillip L. Nico, Ph.D. Clark S. Turner, J.D., Ph.D. Timothy J. Kearns, Ph.D.

Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
email: {pnico,csturner,tkearns}@calpoly.edu

The reports of my death are greatly exaggerated.
—Mark Twain

Abstract

As the field of software development becomes more professionalized and the software development process becomes more formalized, there is a growing belief that the front-line programmer's job is becoming—or must become, out of economic necessity—*de-skilled.*

In this paper we show that essential aspects of the programming activity involve skills that have not been eliminated by progress in the field. The programmer's task does evolve with new tools and methods, but the essential skill—making rational tradeoffs in the presence of an abstract model—remains.

Keywords: Software Engineering, Software Development, De-skill, Programmer

1 Introduction

In recent years the free market has brought about myriad tools to automate various portions of the programmer's job. Witness the development of high-level languages, prototyping tools, formal specification, automated testing, etc. The formalization of the software development process and the professionalization of software developers places software production under the control of a highly-skilled professional engineer. This diminishes the apparent importance of the programmer's job, bringing about calls for lower-paid, less-skilled individuals to perform this simpler task of programming.

In this paper we observe that while the programmer of the 1950s has indeed been eliminated in most cases, the need for the same essential skills has been preserved. We wish to explore the skills essential to computer programming that remain in the face of progress in software engineering.

2 Why does this matter?

The problem of program quality is well known.[1] High defect rates, cost overruns, late schedules are often cited as part of the software crisis. Indeed, poor program quality has been implicated as a contributing cause to death and other injuries in software controlled systems[5].[2] Human errors are part of the programmer's life. At the very least, the programmer might inadvertently fail to faithfully render the design intention of the specifications into code. This can, of course, have major consequences for the behavior of the final system. In other cases, the programmer might even fill in gaps where the specification is either intentionally (a *don't care* case) or unintentionally incomplete (imperfections in a specification). Again, the implication is that a programmer's action can have major consequences for the behavior of the final system.[3]

Software process and, more recently, software engineering have been proposed as part of the solution to the problem of program quality.[4] We do not disagree with the premise that quality processes will lead to improved program quality. Nor do we believe that well educated, highly skilled software engineers are not needed to manage and oversee software projects of modern scale and complexity.[5] The authors do, however, see a danger in the focus of software quality shifting almost entirely to the areas of process

[1] For a good overview with some interesting data from the last decade, see "The CHAOS Report" by the Standish Group (1994) at `http://www.pm2go.com/sample_research/chaos_1994_1.php`

[2] For current examples, see the *Risks Forum* on USENET.

[3] This is one of the unique qualities of software as a "product" in that what appears to be final "assembly" holds the same potential for fundamental change as does the original plan. For software, the programmer could choose to add a flight simulator to a spreadsheet program just because it would be fun to do. This sort of change in final assembly of physical systems is unheard of: the production workers on the Ford Mustang line cannot, one day, decide to build a boat.

[4] As evidenced by SEI's CMM and the ISO 9000 standards for software. Further, the SEI has set forth the SWEBOK, Software Engineering Body of Knowledge, as a standard for software engineering knowledge. See also `http://www.csc.calpoly.edu/~smeldal/SEProgram/documents/NeedSE.html`

[5] In fact, one of the authors is a major proponent of a new B.S. degree program in software engineering at Cal Poly State University. See `http://www.csc.calpoly.edu/~smeldal/SEProgram/` for more information.

and software engineering to the exclusion of the programming activity itself. With the advent of new tools, higher level languages, objected oriented methods, improved processes and software engineers as managers, the programming activity appears to be diminished in value and stature. If a highly paid software engineer is hired to manage a software development process, upper level management might see a chance to hire less skilled (read: cheaper) employees as simple, interchangeable workers at the end of the production line.[6]

3 Essential aspects of the software process

We give our own simplified definitions of the terms for discussion below in Figure 1. We also give a basic model of the software development process in Figure 2. Note that our process model highlights the human activities of development, from "requirements elicitation" to "coding" the program. The automated final translation is shown below that. We intend no fixed points in this model, for we believe there is no objective line where *design* ends and *code* begins, even though these abstractions can be very useful in getting work done. Note also that we do not show the many feedback loops that would be required if we discretized the process into separate, abstract stages. We contend that objective distinctions that point to a line at which point one activity ends and another begins is a mirage[9]. Of course, such distinctions can be useful in dividing up the work and measuring progress in the process[6]. For our purposes, however, the activities of transforming a set of human intentions into a software solution are all part of the "stuff happens." Feedback loops are a first class process activity rather than an abstract way to rationalize a model of artificial stages of development.

As described by Jaffe[3] the process leading from human intention to fully-specified code is one of decreasing abstraction and describing a completely specified solution to the problem presented. Precision is not at issue, highly abstract specs must be precise, but that does not make them adequate to directly implement the code.[7]

Notice that the programmers' task is, considered sequentially, at the "end of the line" before machine implementation. Pressure is on them to produce a working product, to modify the product when needed, to repair known defects. Programmers are expected to deliver the working product for final distribution and sale[8]. They suffer this pressure within the framework of market pressures: limited resources and time deadlines[1]. It is thus natural to ask whether the development process may be adjusted so that the programmer's task requires less skill, is of lower cost,

Programmer:	the last human who interprets an incomplete (ambiguous) specification to produce an unambiguous specification to produce an unambiguous, complete solution with respect to the machine
Program:	a complete, unambiguous specification of a solution to a problem ("final assembly" now *can* be automated.)
Design:	a human decision that produces the specification of a solution to the problem considered.
Programming:	the act of completing the specification of a solution to a problem so it can be run on a machine

Figure 1. Definitions relevant to the software-development process.

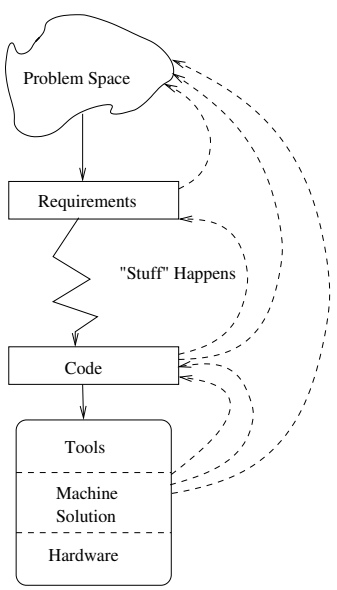

Figure 2. The programmer's place in the development process

[6] Employees who do the bidding of the manager only. We note that if programmers' skills are recognized as valuable and important they may operate with more autonomy as experts in there contribution to software development.

[7] This is often a point of confusion for programming students and those who have little experience with software development. For an interesting discussion, see [2].

or takes less time. We are in the business of automation, after all!

Brooks classifies the difficulties encountered in software development into *accidental* and *essential*[1]. He shows us that there are difficulties in software development that are essential to the activity—they cannot be eliminated. There are also difficulties that are merely accidental—they can be overcome by improved methods and tools. Using these concepts, developers looking for new efficiencies in the development process would have incentives to eliminate the accidental difficulties of programming with tools and methods, to deskill the task as much as is possible. The question is, how much of the programmer's task is accidental and therefore automatable to some extent. Can we reduce the programmer's job to basic translation or transcription? If not, how close can we come?

Were the job of the programmer reduced to a clerical activity, notice the effect on the diagram in Figure 2. The feedback loops to upstream activities and from the downstream machine could not be maintained. The programming activity would become a discrete step in the process with well-defined input from upstream and output downstream. The lack of skills applied to the programming task would not provide a souce of the critical feedback Winton Royce noted when he faulted the waterfall Model[7]. This increases the process's reliance on the foresight of the designers. It also depends on the reliability and expressiveness of the programming language and underlying machine. However, we know that such reliance would be seriously misplaced and is not based on a rational understanding of the programming task[6].

The programmer's job is defined here as taking raw specifications at some level of abstraction and producing another specification (in a programming language) that is sufficiently complete, consistent and unambiguous as to run on a machine. The raw material that the programmer works with (often called "design specification") may have two kinds of incompleteness associated with it, discussed in turn, below.

a. Specifications that are incomplete with respect to the programming language to be used

A simple example of a design specification that is incomplete with respect to the programming language used (C++ or Java) is to specify that an module retrieve an employee record based on a social security number. What is left unspecified is the particular algorithm to be implemented.

That the specifications handed to the programmer are not complete with respect to the programming language is merely to say the solution is not yet specified enough to be implemented on a machine. The programmer must take this specification[8] and write another specification (using a programming language) to satisfy it. Of course, if the specification handed to the programmer *is* complete with respect to the programming language, then a simple translation is all that is needed. However, this step would be easy to automate and such a specification must have precisely the same semantics as the final program in the chosen programming language.

b. Specifications that are incomplete with respect to the problem space.

A simple example where the specifications are incomplete with respect to the problem space is the "program should allow funds to be transferred between checking and savings accounts" What is left out are any constraints on the minimum balance that may affect allowable transfers, etc.

The specifications handed to the programmer are known to be incomplete descriptions respecting the solution to the real world problem at hand[6]. Other than in trivial cases, the complexity of the real world is not amenable to full specification by humans.[9] Thus, software specifications are not expected to be complete when handed to the programmers. The programmers, though, must specify the solution completely enough to implement it on a machine and solve a problem.

4 Evolution of the programmer's task

The task known as computer programming has progressed through many different forms since its invention, but the fundamental task—taking an abstract problem description and mapping it to a series of unambiguous instructions to be performed by the computer—has remained the same. The boundaries of this process have, however, changed over the years. In this section we examine the evolution of the programmer's task along with the programming environment.

With the advent of the stored-program computer in the 1940s came the need for a programmers to teach the system how to do its particular tasks. The tasks in these days were quite simple by today's standards, but so were the tools available. The programmer first had to analyze the problem and develop a solution, then construct a sequence of machine instructions to implement that solution, and, finally, encode those instructions in a form readable by the computing hardware.[10] Programming in this environment was necessarily tedious and error prone and the scope of programs was correspondingly limited, e.g. payroll programs, artillery tables, etc.

With the invention of the assembler, the tedious coding portion of the programming process was eliminated, but the programmer's task remained essentially the same: he had to map the abstract problem into individual hardware

[8] This specification may be at a high level or closer to code. There is no objective standard for "specification."

[9] The essence of the programming activity is problem solving by abstraction, abstraction ignores detail of the real world to achieve enough simplification to create something '"useful enough." For a good thorough discussion, see Jaffe[3].

[10] This is about the only point a computer programmer could properly be called a coder.

instructions. Every action performed by the hardware had to be explicitly specified by the programmer. This required the programmer to have an intimate understanding of the capabilities of the computing hardware and of all its powers and limitations.

High-level languages relaxed the bond between the programmer's specification of a solution and the exact actions performed by the computer. A single FORTRAN statement could produce many machine instructions. The programmer no longer had to understand how a particular machine performed a particular arithmetic or control operation. It was widely believed at the time that high-level languages would eliminate the need for skilled programmers[1, 4]. What actually happened, however, was different. The programmer had a new, more powerful, language in which to specify his solution, but the process of specification remained. In fact, the process became more complex in two different ways: First, the greater capability of the more powerful language led the problem domain to move to larger problems; the language had learned bigger words, but it also had to express bigger thoughts.

The second aspect was even more important: The abstraction provided by the language was not complete. Hardware implementation details such as arithmetic precision crept through. Now the programmer had to not only know about his language but also to recognize "foreign words."

Structured programming, higher-level languages, object-oriented programming, component-based programming, automatic programming, etc., have each, in turn, been heralded as the new development that was going to simplify programming to the point where anybody could quickly and easily produce a solution to a given problem[1]. This belief was misleading, however, because the real effect was to give the programmer a more expressive virtual machine. This virtual machine does eliminate the tedium of specifying the simpler tasks that comprise a program (as did the assembler), but what it leaves behind are the difficult design decisions required in translating the abstract solution to the set of tasks that properly expresses the solution and connecting them in the correct way.

The programming task as we have defined it here involves taking a description of some aspects of the "real world" that we wish to model and specifying it completely so that it is executable on a machine. Making this final step from specification to executable requires linking the a description of a complex (in any interesting problem) situation to a set of components that "simulate" the desired behavior.

The basic underlying problem that is essential to the difficulty of the task of going from specifications to programs is that the systems we are developing are inherently complex (and becoming more so over time.) This means that to solve the programming problem we must link the complex desired behavior of the specification to the set of components offered by any "programming environment."

1. General purpose programming environments provide relatively low level components that can be assembled to model almost any behavior. But this flexibility comes at the cost of either using primitive building blocks or having to learn a large library of functionality. Environments like Visual Studio or all the Java libraries are examples of this complexity

2. Special purpose programming environments can provide a relatively complete set of components but are only suitable for a restricted problem domain. They impose constraints on the types of problem that can be solved and, even within the domain, constraints on how the problem is addressed. Spreadsheets and DBMS are simple examples of this type of programming environment.

In either case above—regardless of the abstraction level—the programmer must do the final translation from the language of the problem space to the language of the solution space. For this translation to be trivial—performable by an unskilled individual—the problem specification must be essentially equivalent to the solution. That is, the problem has already been solved by the specification. What this really means, however, is not that the skilled programmer has been eliminated, but that the virtual machine—the language of the program—has moved again to a higher level. The fundamental process of converting the abstract specification of the problem to the complete specification of the solution remains intact.

A similar effect is seen when the many levels of project management that are now common now are introduced. For a sufficiently complex project, we may have a requirements engineer who interacts with the customer and develops a highly abstract problem specification that is handed off to an architect. The architect, in turn, takes the specification and generates a description of the high-level architecture of the solution. The components of this solution are handed over to software designers, and, ultimately to front-line programmers who produce the ultimate product. At each stage of the process, each of these individuals performs a translation from his own problem language to his own solution language, essentially "programming" the level below. Each stage of the process requires the person performing it to translate an abstract specification into a less-abstract one requiring a skilled translator.

5 Conclusion

The outward appearance of the programmer's job has changed radically over the last half century. Stripped of its external appearance, its essence remains the same: the specification of a solution to a problem that can run on a machine. Tools and methods may change the language of the solution, but do not change the essence of the process through which it is accomplished.

References

[1] BROOKS, F. P. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*, 2nd ed. Addison-Wesley, Reading, Mass., 1995.

[2] JACKSON, M. *Software Requirements and Specification: a lexicon of practice, principles and prejudices*. Addison-Wesley, Wokingham, England, 1995.

[3] JAFFE, M. *Completeness, Robustness, and Safety in Real-Time Software Requirements Specifications: A Logical Positivist Looks at Requirements Engineering*. PhD thesis, University of California, Irvine, 1988.

[4] KRAFT, P. *Programmers and Managers - the Routinization of Computer Programming in the U.S.A.*, 1 ed. Springer, New York, 1977.

[5] LEVESON, N. G., AND TURNER, C. S. An investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (July 1993), 18–41.

[6] PARNAS, D. L., AND CLEMENTS, P. C. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering SE-12*, 2 (1986), 251–7.

[7] ROYCE, W. Managing the development of large software systems: concepts and techniques. In *Proceedings of ICSE 9* (1970).

[8] SCHACH, S. R. *Classical and Object-Oriented Software Engineering*, 3rd ed. McGraw-Hill, 1999.

[9] TURNER, C. S. *Software as Product: The Technical Challenge to Social Notions of Responsibility*. PhD thesis, University of California, Irvine, 1999.