The Essential Similarity and Differences between Mathematical Modeling and Programming

Daniel M. Berry
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
`dberry@uwaterloo.ca`

Abstract

This paper describes mathematical modeling and programming and determines that they are essentially the same activity. However, differences in the way that the products of the two activities are used force them to be done in very different ways with very different goals. A result of these differences is that programming is harder than mathematical modeling, in that the model that the former produces must be more complete than the model that the latter produces.

Keywords: abstract model; exceptions; mathematical modeling; mathematical notation; programming; programming language

Introduction

Some people have wondered about the relationship between mathematical modeling (MMing) and programming. There is clearly some similarity between the two in that they require similar kinds of logical and abstract thinking. There is clearly some difference between the two, if for no other reason than many people who do one do not do the other, although there *are* many exceptions. This paper attempts to describe the similarities and differences between MMing and programming as intellectual exercises.

Essential Similarity

Each of mathematical modeling and programming involves building an abstract model (AM) of a real-world situation or system (RWSS) in some formal language (FL). The resulting mathematical model (MM) or program is the AM. In the case of MMing, the FL is mathematical notation, and in the case of programming, the FL is a programming language, such as Java. Each of these languages is formal because it is defined by humans, each utterance in the language has a precisely defined meaning or is specifically left undefined. In either case, any debate about the AM is about how well the AM models the RWSS and not about what the AM says; everyone who understands the AM understands the same thing.

Essential Differences

In MMing, it is considered acceptable to make simplifying assumptions about the RWSS in order to keep the mathematics tractable[1]. Dealing with the ignored details would cause the mathematics to become unmanagely complex. In other words, MMing really is abstraction in the sense of selective and purposeful ignoring of details. For example in building a MM, one might observe and thus assume that the variation of one quantity is small over the region of interest in order to change a nonlinear problem into a linear problem; one solves the linear problem, knowing that is is valid only in the region of interest.

In programming, the simplifying assumptions that are necessary and acceptable for tractable MMing would yield a program that is unacceptably nonrobust. The program would fail to deal correctly with inputs that the user

[1] If the RWSS is the data of a program, then a tractable model may even venture into unsoundness or inconsistency that can be avoided only at the expense of a complete model of the memory in which the data are stored.

has every right to expect the program to be able to deal with. A program, to be useful, must be able to react in some way to any input that it might be presented, even and especially, if that input were a mistake. If there is an input that a program cannot handle, the program must, at the very least, report to the user that it cannot handle the input that it has been presented.

Thus, for any RWSS, a programmer must be prepared to build an AM that is as complete as possible, even at the cost of highly complex processing and data, i.e., of an intractable model. It is not acceptable to restrict the program's processing to the input that a nice tractable AM can handle.

Tony Hoare once said that "Inside every large program is a small program struggling to get out." While confirming the accuracy of this quotation in recent private communication by e-mail, he added that "the small program can be found inside the large one only by ignoring the exceptions." It is handling these exceptions that makes an otherwise small program a large program and a hairy mess. Even if the central algorithm, the small program, that handles most of the cases is based on the nice tractable MM devised by the mathematician, it will be necessary for the full, big program to deal with the exceptions. Dealing with these exceptions may involve programming the intractable model that the mathematician avoided with the simplifying assumptions that allowed using the nice, tractable MM. However, this division of labor works only if there is an algorithmic way to determine for any input whether or not it falls into the domain of the nice tractable MM. If there is no such algorithm, then the entire large program must be based on the more complete intractable MM that the mathematician prefers to avoid.

Note that it is the incomplete modeling of the RWSS by the AM that creates and determines the exceptions and not the other way around. Each AM, clean and simple, or not, has its own exceptions determined by the portions of the RWSS that it does not model. In fact, any AM of any but the simplest, totally man-made RWSS models the RWSS incompletely [2]. Therefore, having to deal with exceptions in programming is unavoidable.

So, it is more like that around every nice, clean, small program that does the job 90% of the time is a large, hairy, messy program that does the job in the remaining 10% of the time, The small program is about 1/10 of the size of the large program. This ratio is approximately the same as Fred Brooks's 1/9 ratio between the size of a program and the size of the software system product (SSP) [1] that contains the program. The SSP does all the extra stuff that makes the program usable by more than the program's authors and as part of a complete real-world system.

Perhaps it was the slow 10-year discovery of the large TEX program [4] containing the heart of TEX's formatter, the really neat algorithm [3] that breaks the input text into aesthetic paragraphs of left- and right-justified lines that prompted Don Knuth to say [5],

> What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD.... From now on I shall have significantly greater respect for every successful software tool that I encounter. During the past decade I was surprised to learn that the writing of programs for TEX and for METAFONT proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks.

His writing of a publically usable, relatively stable version of TEX had apparently taken about 10 times longer than he had expected[2].

There is occasionally a misplaced concern for space or time efficiency in the development of software for dealing with a complex and changing RWSS. Sometimes, what needs to be optimized is not computational space or time, but program simplicity, in order to make the program's initial development or future, inevitable modification less error prone. Often a space or time optimal algorithm is significantly harder to understand, to program, and to modify than a simple, brute-force algorithm for the same function. Moreover, for many a program, with current computer memories and speeds, a simple brute-force, but relatively slow algorithm works just fine in practice. Thus,

[2] The author remembers hearing a talk by Knuth in 1979 in which he said that he had hoped to be able to demonstrate a running version of a typesetting program on which he had been working for about a year, but the software was not finished yet. Then in 1989, Knuth gave a keynote lecture in which he talked about the typesetting program on which he had been working over the past decade or so [5].

developing algorithms for use in running software is different from developing optimal algorithms that are the objects of study in the fields of algorithmics and computational complexity.

It is not the intention of this note to deprecate the value of MMing, particularly that done in the context of programming to find a highly effective central algorithm for the program being developed. Without this MM-inspired central algorithm, every input would be an exception. Tony Hoare suggested an everyday example of the importance of this central AM: A large part of the grammar of any natural language is taken up by irregular verbs. Without a model of what is regular in verbs, every verb would be irregular[3]

Conclusion

Each of MMing and programming involves building an AM of some RWSS. The differences between the two lay in the degree to which approximations, simplifying assumptions, and incomplete modeling of the RWSS is acceptable. In MMing, tractability often takes precedence over completeness of the modeling, while in programming, completeness of the modeling is usually absolutely essential to the program's being usably correct. Moreover, in programming, optimizing a program's readability often takes precedence over optimizing its space or time usage.

Acknowledgments

The author thanks George Labahn for his answers to the author's questions about the kind of MMing that he does. He thanks Tony Hoare for confirming a quotation attributed to him, for comments on a previous draft of this paper, for some ideas for improving and strengthening the paper, and for an example used in the paper. Finally, the author thanks Nancy Day and Richard Trefler for comments on previous drafts of this paper.

References

[1] Brooks, F.P., Jr., *The Mythical Man-Month: Essays on Software Engineering*, Second Edition, Addison Wesley, Reading, MA (1995).

[2] Jackson, M., *Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices*, Addison-Wesley, London, UK (1995).

[3] Knuth, D.E. and Plass, M.F., "Breaking Paragraphs into Lines", *Software—Practice and Experience* **11**, pp. 1119–1184 (1981).

[4] Knuth, D.E., *Computers & Typesetting, Volume B: TEX: The Program*, Addison Wesley, Reading, MA (1986).

[5] Knuth, D.E., "Theory and Practice", Keynote address at IFIP Congress 1989, Report No. STAN-CS-89-1284, Computer Science Department, Stanford University, Palo Alto, CA (1989).

[3] Interestingly, this everyday example is not far off from being a programming example. this author supervised the development of a program one of whose requirements is to identify plural nouns with recall as close as possible to 100%, even at the expense of high imprecision. While imprecision can be quite high, 100% imprecision is really not very helpful to the user. It came down to a tradeoff between (1) an ever growing list of all known plural nouns, regular and irregular, currently about 12K of them, with a very simple text-matching algorithm and (2) an algorithm embodying general rules, with a much smaller exceptions list of only irregular nouns. The first approach is the all-exception approach, and the general rules of second approach suffer from both poor recall and poor precision.