THE CREATION OF SYSTEMS PROGRAMMERS

Professor W. C. Lynch

Case Western Reserve University/
Computing Laboratory,
University of Newcastle upon Tyne,
Newcastle upon Tyne.

Abstract:

This paper examines the relationship of Computer Science to the art of programming. It pays particular attention to the question of what role the art of programming should play within the University and to the steps which the University should take in order to implement properly that role.

1. Introduction

Programming, particularly systems programming, is an art, excellence in which is aspired to by many and achieved by few. The processes and mechanisms by which these artists are created and the relationship to computer science and computer scientists is a topic which I wish to pursue in this paper.

2. The role in computer science

Let us begin rather obliquely by examining computer science. If the name computer science is to mean anything, this discipline must first and foremost be a science. This is to say it must investigate, by theory and by experiment, a body of organisation or mechanism that is more or less beyond its direct control. Physics and chemistry obviously qualify as sciences; social science qualifies on the grounds that it investigates a mechanism, human activity, which is presumed to be beyond the direct control of the social scientists; engineering science is justified as a science on the grounds that it investigates the art of engineering practice and attempts to reduce it, via scientific methods, to a more quantitative discipline. It follows then that computer science should investigate, via theoretical models, observation, and experimentation, the practice of applying computers to their wide range of application.

Computer science functions by drawing upon the successes of past theory and upon the experiments and observations of the art of computer programming, in order to formulate new models describing the processes by which programming is practiced. Such models must then be tested for accuracy and relevance against the results of the actual practice of programming. Those models which prove useful will then, hopefully, have some impact upon the practice of the art.

The computer scientist is then heavily dependent upon the practice of computer programming in the hands of the programming artists for the raw material upon which his science is built. A scientist without either the scientific method or the material to investigate with it is not a scientist at all.

3. The art of computer programming

Programming then is an art, not a science. Common experience with programming and programmers tells us that it contains a great many

elements which defy scientific description. One can speak meaningfully about technique, style, and aesthetics; the relationship of these to utility is roughly the same as it is in many other areas which involve both artistic and utilitarian aspects, architecture for example.

If programming is an art, we may then investigate the creation of programmers as artists rather than as scientists. A good way to begin such an investigation would be by examining the creation of artists in other areas.

3.1 Theory and technique

One important element in the creation of an artist in any area is the learning of the relevant theory and technique which have been developed through scientific investigation of the art form. In painting, for example, this would involve the geometry of perspective, technique of painting materials and the techniques involved in the physical application of these materials. In programming, the theory and technique involve the acquisition of those basic principles which, through computer science, have been found to be useful. A roster of courses for most computer science departments will indicate specifically the items which are included under the heading of theory and technique.

3.2 Environment, past and present

A vital ingredient for any artist is the environment, past and present, in which he finds himself. A writer cannot create significant literature without drawing from his environment significant topics on which to write. A programmer, operating in the business sphere, cannot construct meaningful systems and programs without having significant experience with the business area in which he is dealing. A scientific programmer certainly has significant difficulties if he has very little familiarity with the scientific area in which he is constructing the application programs. The artist's past environment influences to a very great extent the efficacy with which he is able to operate. The artist also draws upon his past environment and experience in another way — the creative artist will certainly have had exposure to important past works by other people. Close study of significant works leads to an understanding of the techniques and style previously employed and certainly generates new ideas and new thought concerning future works. An artist's development and ability to function is also dependent on his present environment in the sense that he must be given sufficient latitude in order to innovate and express style and its

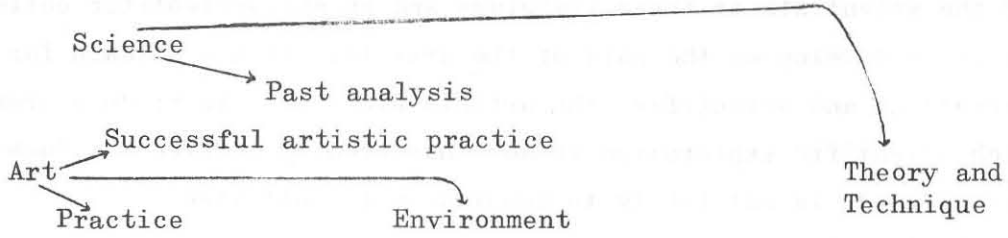
aesthetics. A project with too rigid a specification will be carried out in a mechanical rather than artistic fashion. The end result is apt to bear the mark of this and the artist will certainly profit very little by the experience.

3.3 Clinical Practice

A third major facet in the creation of an artist is the opportunity for practice and critique. One would certainly have no chance of creating an accomplished artist, writer or musician without giving them sufficient practice in their art, together with ample critiques of their style and accomplishment. This clinical experience occupies a major portion of the time spent in the training in the arts.

4. Science vs. Art

The aspects that I have dwelt on are then summarised in Figure 1.



The science area at the top depends for raw material upon the past analyses created by previous scientists and upon the current successful artistic practice as determined by the contemporary artists. The results of the scientific endeavour then provide new analyses upon which science can be partially based and provide new theory and technique which can be applied to the art. The artists clinical practice then depend upon the theory and technique provided by the science, and upon the environment and experience which is provided primarily by the actual practice of the art. The practice of the art provides the endeavour which the science the proceeds to investigate.

It may strike the reader that the above comments are so obvious to be hardly worth stating. It is worthwhile to contrast the above principles with some of the current states of affairs of Computer Science in universities. Let us imagine for a moment that the interaction between the science and the art is broken. The interaction paths in Figure 1 will be erased. We may then easily predict the results of such an isolation; first of all the scientists will, on the basis of past analysis, continue to propound theroies which will not be subjected to experimentation. There

will thus be no way to determine the truth, validity, or relevance of the theory. The scientists will then be in the same position as mediaeval scientists who believed it beneath their dignity to attempt an experiment to determine whether or not their theory was valid. An even more serious and subtle difficulty also practiced by the mediaeval scientists is the tendancy to propound theories which predict everything and for which no experiment can be devised to disprove the theory. Such a theory fails the test of Occam's razor, and is useless on the ground that it says nothing. Such a science will wander rapidly away from reality and provide no help whatsoever for the practice of the art. Once this has happened, the scientists will have little to say in the way of theory, the technique of which is of use to the artists. The artists in turn will have little use for the scientists or their trappings and an anti-scientific outlook is very likely to develop on the part of the artists. With a disdain for things theoretical and scientific, the artists are likely to produce creations in which scientific exploration is more than usually difficult. Such a separation once created, is not likely to spontaneously heal itself.

5. Implications

5.1 Computer Science Department vs. Computation Centre

Let us now try and identify these components within the typical university. Computer Science is centred within the Computer Science Department and its graduate programmes. The programming art is generally located within the Computing Centre and to some extent, within the undergraduate curriculum. In some cases, the Computer Science Department is estranged from the Computing Centre with virtually no communication between the two organisations. This has the obvious effect of almost totally cutting the interaction between science and art, leading to the above mentioned problems. The Computer Science goes off in some direction which is not particularly useful. A little more subtle is the effect upon the actual practice of programming within the computation centre itself. Without theorectical guidance, such centres are apt to dissipate their resources in a fruitless frontal attack upon massive programming problems.

One might expect the science and art to meet in the undergraduate curriculum. A perusal of many undergraduate curricula will reveal that they are usually woefully deficient in terms of providing an environment for the growth of the student and in terms of providing significant practice and

criticism. Such a curriculum is apt to be so full of theory and technique that there are very few course hours left for other environmental subjects contributing to the student's education. As I commented earlier, the artist must have some background dealing with the rest of the world that he is to operate with. It is likely that it is much more important that he be educated in physics or business or any of a number of other areas than it is for him to be up on the latest form of tree processing.

5.2 Practice of the Art - Personal Experience

Advanced practice of the art occupies so much time and effort that this aspect is either given little time or eliminated completely from the curriculum. Of course, short exercises are always included in any computer science curriculum. Many curricula however, present insufficient opportunity for the student to design, undertake and construct significant programming project in a form which lends itself to significant critique. Such major efforts are usually restricted to individual study projects where the opportunity for criticism, particularly from other students, is quite limited. Many curricula are therefore long on techniques and the latest programming fads, and short on the clinical aspects which lead to growth in the art.

This practice at Case Western Reserve University is implemented by embedding a term project within the second semester of the systems programming course. This course is a two semester sequence, the first semester of which is concerned with the specific techniques of systems programming, such as symbol table techniques, scanning techniques, parsing techniques, code generation techniques and list processing techniques. A number of relatively short exercises to be run on the computer are included within this first semester material. The second semester material deals with two quite different topics - the first, and perhaps the most important aspect of the course, is the project which has been referred to whereby each student constructs and operates a compiler for an algebraic language. The lecture material of the course deals with something quite different, the topic of context and operating systems.

The sixty students within the course are divided up into teams of two. They are given the complete specification of a compiler to be produced. The language of this compiler is called Rational and it is an algebraic language which deals with arithmetic variables of types rational, integer and boolean. A normal complement of computation and jumping operators are included. The language contains no block structure, and only the most primitive procedure calls. Both the language and the output to be produced

are rather rigidly specified. This rigid specification has been imposed after much past experience so that the students may compare notes on their implementation and criticise each other's efforts. This rigidness also aides in providing a meaningful grade and meaningful critique from the course instructors. Such critique, as I have indicated previously, is most important.

Within these rigid guidelines, the students are allowed a great deal of freedom. They are free to choose their implementation language and students have chosen, from time to time, assembly language, Algol 60, FORTRAN, and COBOL. They are advised that it will be easiest in Algol 60, and in fact, most students do choose this route. The teams of two are also free to divide the effort in any way they see fit. In this way they are taught the value of teamwork and also the difficulties of communications that arise when more than one person works on a project. This also allows them a more intimate way of reviewing and criticising each other's work. The division of effort is usually along the lines of one partner taking the scanner and parser and the other partner taking the code generator. Neither one of them realises at the outset that the bulk of the coding effort is involved in producing the output and particularly the listings of object code. As a result this is about a $\frac{2}{3}/\frac{1}{3}$ division of labour. This in itself, provides the students with a valuable lesson.

Of the thirty teams that initially begin the project, we find that perhaps 26 or 27 of them will successfully complete it, obtaining a compiler which produces code which operates more or less correctly. The brighter students will go beyond the specification and will build translators which will accomplish much more than is required. Usually the best four or five teams in the course will attempt global optimisation within the compilers, usually with a great deal of success. The code will use a multiplicity of registers and will retain register contents across statements. It is usually the case in such beginning efforts that these globally optimised compilers do a less than perfect job of local optimisation so that there is still a good deal of room for improvement. The test deck applied by the instructors contains the usual repertoire of extremely nasty examples. Most of the compilers have at least several blatant errors in them resulting from the instructors applying cases which they had not thought of. This grading procedure which is possible because of the standard and rigid specification is itself a revelation to the students.

This project has been on-going for approximately ten years at Cas Western Reserve, and the format of the project has evolved over the years to the one described. There is thus a very large backlog of practical experience indicating that this is an appropriate format for balancing the various factors involved in the practice of the programming art.

5.3 Management vs. Structure

One might expect these aspects to be supplied after graduation in the form of employment experience. On the university campus, this would mean within the computation centre. In such circumstances, computation centres particularly take on an aspect of education; if in fact such clinical education is sponsored and fostered by the computation centre, particularly in conjunction with the computer sciences, a very pleasing organisation can result. In many cases, however, the centre functions in a very goal oriented way leading to projects which are very highly structured and organised and therefore constrained in such a way as to not leave much room for creativity or development of style. Such a short sighted attitude would be a computation centre's contribution to its estrangement with the computer sciences.

A theorem due to Dr. Melvyn Conway is here relevant concerning the structuring of programs and programmers. A statement of his theorem is as follows:

Theorem

The structure of a program (as a graph) is isomorphic to the organisation chart (as a graph) of the organisation producing the program.

Conway proves this theorem as follows: First of all, he argues that the organisation chart is a homormorphic image of the structure of the program being produced. This is due to the fact that each module of the program must be assigned as an entity to a single subdivision within the corporation. If one attempts to assign a module to more than one subdivision within corporation, they will themselves have to devise a communication path between halves of the module so as to be able to divide the effort. The program will then acquire additional structure reflecting the division across the organisation producing it. It may, of course, at this stage, be possible for a single organisation entity to implement more than one module of the program. Thus, Conway has established the homomorphism from the program structure to the organisation structure. Conway then extends this homormorphism to an isomorphism by an application of Parkinson's Law.

If an organisation is to design and construct programs in a rational way, it then follows that the organisation must adapt its structure to the program rather than the other way around. This means that the concept of mobility is a highly important one for programming organisations. We then infer that an enlightened organisation must take into account the maintenance of a creative environment in the design and organisation of its programs, and hence of the organisation of itself. The organisation of, for example, the computation centre must be mobile enough to accommodate itself to these principles.

If several systems are under development simultaneously, we infer that the organisation producing them must itself be organised along several different pathways simultaneously. Such seemingly amorphous organisations are typical of very small companies and in many ways account for their spectacular success. With such a small company or computation centre, one far-sighted manager can keep in his mind all the various organisational subtleties involved in the production of the various systems he is concerned with. He can also keep in mind development of the appropriate environment for his programmers. Retaining such mobility in medium-sized organisations is a challenging and largely unsolved problem. It is obvious that the computation centres of many universities are in a position to make significant contributions in this area.

6. Conclusion

Ideally then we have the computer sciences and the programming arts in a symbiotic relationship, each benefitting from the developments in the other; the programming art benefits from the relevant theories and techniques developed by the computer scientists, and the computer scientists using the results and the systems proved in practice as basic material upon which to build relevant models. The strength or weaknesses of the products of the universities will depend in large measure as to how well or how poorly this relationship has been established within the university. A poor relationship will lead to a narrow-minded and inferior product; an excellent relationship and an excellent environment will lead to graduates who are able to function effectively in a broad spectrum of areas and whose education will carry them over a considerable time span.

We have then reviewed the relationship of computer science to the art of programming. We have examined some of the ramifications and impacts of this relationship upon the development of computer science and computer

scientists. We have investigated some specific ways by which the practice of the art of programming has been implemented in the past and we have examined some of the philosophical issues with which university computation centres are or ought to be concerned with respect to the relationship of computer science to the art of programming.

Questions and Answers

Professor Dr. H. Wedekind of the Technische Hochschule Darmstadt raised the question early in the talk as to whether programming could be treated as an art. He expressed the point of view that since an end product of some economic values was involved, the technical and professional considerations could not be ignored in favour of an aesthetic approach. Professor Lynch replied that he was in complete agreement with the point of view expressed by Professor Wedekind but as the previous speakers in the Symposium had dealt extensively with the scientific and professional aspects of programming, he wished to concentrate upon the artistic aspects arguing that the art of programming beyond that encompassed by science and professionalism was not a null object. As a result, Professor Lynch stated that his talk would be very heavily biased towards the artistic aspects of programming which as yet had not been dealt with by the Symposium.

During a discussion of the compiler project, Professor Sidney Michaelson of the University of Edinburgh inquired as to what level were the students involved in the programming course. Professor Lynch replied that about $\frac{2}{3}$ of the students were advanced undergraduates, and $\frac{1}{3}$ of the students were beginning graudate students. For both of these classes of students the systems programming course was perhaps the third programming course that they had had.