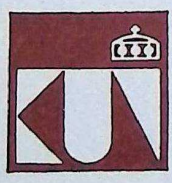
SELECTED ANNOTATED BIBLIOGRAPHY ON SOFTWARE ENGINEERING

Karl Kleine



Informatics Department
Faculty of Science
Nijmegen University, The Netherlands

Selected annotated bibliography on Software Engineering

Karl Kleine (*)

September 77

Katholieke Universiteit Nijmegen
Faculteit der Wiskunde en Natuurwetenschappen
Afdeling Informatica
Toernooiveld, Nijmegen, The Netherlands

(*) Fellow of the Faculty of Science of the KUN

This bibliography tries to be an introductory guide to the literature on "Software Engineering". This term has been in use for about 7 years, but there is no really satifying definition of what it's really about. We will take a more or less personal view and use the following classification:

I Books on software engineering, survey articles

A General

II Management of software projects

B Organization of software projects

C Costs and schedules estimation for software development

D Human aspects

III Technical issues

E Design issues

F Modularity, program decomposition schemes

G Specifications

H Programming methodology and 'Structured Programming'

I Program proving and verification

J Testing and debugging

K Programming errors and their causes

L Software production tools and aids

M Programming languages

N Documentation

O Portability and adaptability

P Efficiency, benchmarking and optimization

IV Miscellaneous

Q Curricula for software engineering education

R General references

The criteria used for the selection were:

a) The main aspects listed above must be covered.

b) Significance and technical value.
This is of course a question of personal judgment of the reviewer.

c) Restriction to 5 - 10 references per subject.
More would not be useful to the ordinary reader.

d) References to bibliographies, surveys and tutorials were included whenever available.

e) All items must be easily available.
This rules out a number of internal project reports, theses, industrial documentation, etc.. The references cited here are either a (chapter of a) book, an article in a technical journal or contained in some conference proceedings.

In the following, each section is preceeded by a not too long review stating significance and subjects covered, and giving some recommendations for the uninitiated reader.

Acknowledgements:

This bibliography originates from courses / seminars at Berlin Technical University given by C.H.A.Koster, W.Koch, W.Simonsmeier. P.Holager reviewed drafts and provided useful comments.

Opportunity to (re-)read the material and prepare this survey was granted by the Faculty of Science of the Katholic University of Nijmegen by a fellowship to the author.

The term 'Software Engineering' was apparently coined by the title of the NATO conference at Garmisch, Oct.68. The proceedings of the conference [A1] and its follower [A2] mark the beginning awareness of the problems in the real production of large software systems. For this reason these books have their definite place in the history of computer science, and are still worth reading today. The material presented is somewhat dated, and extensively rephrased elsewhere, but it does give a good impression on the problems encountered in the production of software.

In spring 72 an "Advanced course on Software Engineering" was held at Munich Technical University. The material of that course [A3] gives a fairly good, quite broad survey on software engineering. A more recent publication of approximately the same scope is [A6]. Parts of both books are referred to throughout this bibliography.

[A4, A5] are short survey papers. While [A4] is more a 'keywords and buzzwords' explanation paper, [A5] is a broad, yet concise narrative of the problems and approaches of software construction.

Conference proceedings of high interest are [A7, A8, A9]. They all contain a large number of important articles and are good sources for further reference.

Recommended:
as short survey (e.g. start-up article for students): A5,
for a course on software engineering: A6, A3, A4

[A1] Ed: P.Naur, B.Randell,
Software Engineering,
Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th Oct. 68,
NATO Scientific Affairs Division, Brussels, Jan.69

[A2] Ed: Buxton, B.Randell,
Software Engineering Techniques,
Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31th Oct. 69,
NATO Scientific Affairs Division, Brussels

([A1, A2] are not easy available; a combined reprint has been announced by Petrocelli/Charter publishers.)

[A3] Ed: F.L.Bauer,
Software Engineering, An advanced course,
LNCS 30, Springer Verlag, 75,
(formerly Lecture Notes in Economics and Mathematical Systems 81, Springer, 73)

[A4] D.T.Ross, J.B.Goodenough, C.A.Irvine,
Software Engineering: Process, Principles, and Goals,
Computer, vol.8#5, May 75

[A5] B.W.Boehm,
Software Engineering,
IEEE Transactions on computers, vol.C-25#12, Dec.76,
pp.1226-1241

[A6] Ed: E.Horowitz,
Practical strategies for developing large software systems,
Addison-Wesley, 75

[A7] International conference on reliable software,
Los Angeles, April 75,
SIGPLAN Notices, vol.10#6, June 75
also: IEEE Cat.No. 75CH0940-7CSR

[A8] 2nd international conference on software engineering,
San Francisco, 76
IEEE Cat.No. 76CH1126-4C

[A9] Ed: P.G.Hibbard, S.A.Schuman,
Constructing quality software,
proc. IFIP-TC2 conference, Novosibirsk, May 77
North-Holland, to be published

By now there is a slowly growing literature on the subject, but we still do not know much about the subject. There exists, however, a large literature about project management in general, which should not be disregarded. Software does not seem to be that different, in most respects.

Most papers only descibe the phases a project goes through and give some comments on the management needed. Examples are [B1, B2, A5]. They rely more or less on classical management techniques.

For the organization of the teams actually designing and producing the code, there are two new models, the chief programmer team and the software family. The former is described in [B4, B3] and related to design and coding techniques in [B5]. The software family is descibed in [D2, D1].

[B6] presents a database system for management support providing services like status recording and surveys.

Recommended: B3

[B1] D.Tsichritzis,
Project Management,
in: A3, pp.374-384

[B2] E.B.Daly,
Management of software development,
IEEE Transactions on Software Engineering, vol.SE-3#2,
May 77, pp.229-242

[B3] F.P.Brooks,
The mythical man-month,
Addison-Wesley, 75

[B4] F.T.Baker,
Chief programmer team management of production programming,
IBM system journal, vol.11#1, 72

[B5] F.T.Baker,
Structured programming in a production programming environment,
in: A7, pp.172-183
reprint: IEEE Transactions on Software Engineering,
vol.SE-1#2, June 75, pp.241-252

[B6] H.Bratman,
Automated techniques for project management and control,
in: A6, pp.193-211

[B7] special issue on management of software projects,
Datamation, Dec.74
(contains excerpt from B3)

In the last few years some papers have appreared that discuss the effort needed for the production of large software systems. Surveys for smaller systems are still missing. The most widely known data concerns IBM's OS/360, US military projects as SAGE, and the NASA space flights. The latter both suffer somewhat in generality because of peculiarities in US-government contracts.

[C1] presents some of the conclusions of an USAF study on their computing facilities requirements for the 1980ties. It gives a fairly good presentation of the problems and costs envisaged. [C2] is the most complete source generally available at the moment. Its size and large number of figures makes it somewhat hard to read, and so [C3] is a welcomed condensed version of it, well suited as course material. [C4] and [C5] treat manpower planning. [C4] does this on the basis of a 30-20-50 rule (duration of design, code, test phases) and complexity / interaction estimates for subsystems. [C5] looks at lines of code produced and its relation to programmer ability, problem area and tools (mainly programming languages) used.

Recommended: C3, C1.

[C1] B.W.Boehm,
Software and its impact: A quantitative assessment,
Datamation, May 73, pp.48-59

[C2] R.W.Wolverton,
The cost of developing large-scale software,
IEEE Transactions on computers, vol.C-23#6, June 74,
pp.615-636

[C3] R.W.Wolverton,
The cost of developing large-scale software,
in: A6, pp.73-100,
(edited & compacted version of C2)

[C4] J.D.Aron
Estimating resources for large programming systems,
in: A2, pp.68-79

[C5] J.R.Johnson,
A working measure for productivity,
Datamation, Feb.77, pp.106-110

The fact that software designers and programmers are humans in a social environment has largely been neglected in the literature. We have only found the three references given below, the team structuring proposal known as 'chief programmer team' [B4], which emphasizes the organizational side, and some recent research in error proneness of e.g. features of programming languages (c.f. section K on programming errors and their causes).

[D1] can only be characterized as outstanding, stressing the human's part, and not the machine's. Weinberg cares for programmer's attitudes and their relation to their job as well as to the other members of their programming team. He advocates 'egoless' team programming. [D2] is a continuation of [D1] about team formation and structure and its effects on a project.

[D3] reveals an important observation about the structure of systems, known as 'Conway's Law': The structure of a (software) system is congruent to the design team structure.

Recommended: D1, D3

[D1] G.M.Weinberg,
The psychology of computer programming,
van Nostrand Reinhold Co., 71

[D2] D.Freedman, D.Gouse, G.M.Weinberg,
Organizing and training for a new software development project - that big first step,
National Computer Conference 77, Dallas, AFIPS vol.46, pp.255-259

[D3] M.E.Conway,
How do committees invent ?
Datamation, April 68, pp.28-31

Figures from [E1] show that 64% of the errors found in software during and after delivery result from the design phase, and only 36% from the coding. This drastic figure demonstrates the importance as well as the common neglection of that early phase.

All tasks of the software have to be identified in an requirements analysis [E7, E8] and subsystems and modules must be defined and specified. Modules are treated in more detail in sections F and G. In this section we are interested in the way to get to an internal program structure composed of subsystems / modules, and how these modules form together a coherent system. The two viewpoints of outer and inner appearance are often not clearly separated. In most cases only the internal program structure is discussed in the literature.

Of special interest is the amount of information about other modules needed for the design and coding of a particular module. Parnas discusses this aspect in several articles [E5, F1, G1] and advocates the 'information hiding principle', separating the implementation of a module / subsystem from its outer appearance.

[E6, E7, E9] survey special techniques and tools aiding the system design.

Recommended: E5, E1, E6

[E1] B.W.Boehm,
Software design and structuring,
in: A6, pp.103-128

[E2] P.G.Neumann,
Toward a methodology for designing large systems and verifying their properties,
4.GI-Jahrestagung, Berlin, Oct.74,
LNCS 26, Springer 74, pp.52-65

[E3] W.P.Stevens, G.J.Myers, L.L.Constantine,
Structured design,
IBM systems journal, 74, pp.115-139

[E4] B.Liskov,
A design methodology for reliable software systems,
FJCC 72, AFIPS vol.41, pp.191-199

[E5] D.L.Parnas,
Information distribution aspects of design methodology,
IFIP 71, TA-3, pp.26-30

[E6] B.Shneiderman,
A review of design techniques for programs and data,
SOFTWARE Practice and Experience, vol.6, 76, pp.555-567

[E7] D.T.Ross, K.E.Schoman jr.,
Structured analysis for requirements specification,
IEEE Transactions on Software Engineering, vol.SE-3#1,
Jan.77, pp.6-15

[E8] D.T.Ross,
Structured analysis (SA): A language to communicate ideas,
IEEE Transactions on Software Engineering, vol.SE-3#1,
Jan.77, pp.16-34

[E9] J.F.Stay,
HIPO and integrated design,
IBM system journal, 76, pp.143-154

Decomposition of a program into modules serves to reduce complexity. (We will here disregard decomposition forced by machine limitations.)

Parnas [F1, E5, G1] advocates the minimization of interfaces between modules as criterium for a "good" modularization.

The hierarchical ordering of modules is discussed in [F2, F3, F4]. Hierarchy imposes restrictions on the define-use relation between modules. As each module defines new functions using available old ones, this may lead to levels of abstraction, c.f. [E4].

[F5] discusses the connection of modules, abstraction and data type definition.

Recommended: F1, F3

[F1] D.L.Parnas,
On the criteria to be used in decomposing systems into modules,
CACM, vol.15#12, Dec.72, pp.1053-1058

[F2] G.Goos,
Hierarchies,
in: A3, pp.29-46

[F3] E.W.Dijkstra,
The structure of the "THE" multiprogramming system,
CACM, vol.11#5, May 68, pp.341-346

[F4] O.J.Dahl, C.A.R.Hoare,
Hierarchical program structures,
in: Dahl, Dijkstra, Hoare: Structured programming,
Academic Press, 72

[F5] C.H.A.Koster,
Visibility and types,
Conference on data, Salt Lake City, 76,
SIGPLAN notices, vol.11, special issue

After having defined an overall structure of modules forming a program, as discussed in sections E and F, we deal with the specification of the modules and their connections to each other.

[G1] is a continuation of [E5, F1] now dealing with the details of specification.

[G2] introduces a module interconnection language (MIL) to describe the overall structure.

[G3, G4, G5] treat the specification of abstract data types. The authors connect the module concept strongly with the concept of abstract data types.

Recommended: G2, G3, G1

[G1] D.L.Parnas,
A technique for software module specification with examples,
CACM, vol.15#5, May 72, pp.330-336

[G2] F.L.deRemer, H.Kron,
Programming-in-the-large versus programming-in-the-small,
in: A7, pp.114-121
reprint: IEEE Transactions on Software Engineering,
vol.SE-2#2, June 76, pp.80-86

[G3] B.Liskov, S.Zilles,
Specification techniques for data abstractions,
in: A7, pp.72-87
reprint: IEEE Transactions on Software Engineering,
vol.SE-1#1, March 75, pp.7-19

[G4] B.Liskov, S.Zilles,
Programming with abstract data types,
SIGPLAN Notices, vol.9#4, April 74, pp.50-59

[G5] J.Guttag,
Abstract data types and the development of data structures,
Conference on Data, Salt lake City, 76
CACM, vol.20#6, June 77, pp.396-404

Starting around '68 there has been a continuing controversy about 'structured programming'. The contributions can be classified as:

- Maxims of the great gurus of programming methodology, as Dijkstra, Hoare, Wirth [H1, H2, H3, H4, H5, H6].
- The 'goto-debate', triggered by [H10] (though the issue was already raised much earlier in [H11]), leading to such statements as [H12] (compare answer [H13]). The issue is now hopefully settled by [H14].
- The practioneers, who learned to survive in an imperfect, even hostile world by imposing standards on their work [H15, H16, H17], or by incrementally modifying their programming language to make it "structured" [H18, H19].
- and finally people trying to provide some general guideline [H20, H21, H22], last not least for their own benefit, as 'SP' sells on the market [H8, H9]. (Jackson and Yourdon both run software consultancy companies).

Recommended: H7, H3, H15

[H1] N.Wirth,
Program development by stepwise refinement,
CACM, vol.14#4, April 71, pp.221-227

[H2] P.Naur,
An experiment on program development,
BIT, vol.12, 72, pp.347-365

[H3] O.J.Dahl, E.W.Dijkstra, C.A.R.Hoare,
Structured programming,
Academic Press, 72

[H4] E.W.Dijkstra,
A discipline of programming,
Prentice-Hall, 76

[H5] N.Wirth,
Systematic programming,
Prentice-Hall, 73
german edition: Systematisches Programmieren,
Teubner Studienbuecher, 72

[H6] N.Wirth,
Algorithms + Data = Program,
Prentice-Hall, 76
german edition: Algorithmen und Datenstrukturen,
Teubner Studienbuecher, 75

[H7] special issue on programming,
Computing Surveys, vol.6#4, Dec.74

[H8] special issue on 'structured programming',
COMPUTER, vol.8#6, June 75

[H9] special issue on 'structured programming',
Datamation, Dec.73

[H10] E.W.Dijkstra,
Goto statement considered harmful,
CACM, vol.11#3, March 68, (letter to the editor)

[H11] P.Naur,
Algol programming: goto statements and good algol style,
BIT, vol.3#3, 63, pp.204-208

[H12] S.W.Smoliar,
On structured programming,
CACM, vol.17#5, May 74, pp.294 (ACM forum)

[H13] D.Gries,
On structured programming - a reply to Smoliar,
CACM, vol.17#11, Nov.74, pp.655-657 (ACM forum)

[H14] D.E.Knuth,
Structured programming with goto statements,
in: H7, pp.261-301

[H15] B.W.Kerningham, P.J.Plauger,
The elements of programming style,
McGraw-Hill, 74
(excerpt in H7, pp.303-319)

[H16] C.Floyd,
Strukturierte Programmierung fuer COBOL - Anwender,
Hoffmann & Campe, 74
(in german)

[H17] A.van Gelder,
Structured programming in COBOL: An approach for
application programmers,
CACM, vol.20#1, Jan.77, pp.2-12

[H18] L.P.Meissner,
On extending FORTRAN control structures to facilitate
structured programming,
SIGPLAN Notices, vol.10#9, Sept.75, pp.19-30

[H19] D.J.Reifer,
The structured FORTRAN dilemma,
SIGPLAN Notices, vol.11#2, Feb.76, pp.30-32

[H20] J.D.Aron,
The program development process,
part I: The individual programmer,
Addison-Wesley, 74

[H21] M.A.Jackson,
Principles of program design,
Academic Press, 75

[H22] E.Yourdon,
Techniques of program structure and design,
Prentice-Hall, 75

The hope that it might become practical to produce program units with a mathematical proof of correctness, was imposed on the computing community by Hoare in '69 in [I4].

A good state-of-the-art survey is given in [I1] (somewhat older: [I2]). [I3] is a tutorial.

Verification by symbolic execution is another approach to this treated in section J.

There is intense research activity in this area, even leading to axiomatic language definition [I6] and incorporation of exrensive facilities for verification in programming languages [I7].

However, up to now there are no practical results for real-life software. A well readable critique of the approach is [I8].

Recommended: I1, I8

[I1] R.L.London,
A view of program verification,
in: A7, pp.534-533

[I2] B.Elpass, K.N.Levitt, J.Waldinger, A.Waksman,
An assessment of techniques for proving program correctness,
Computing Surveys, vol.4#2, June 72, pp.97-147

[I3] S.L.Hantler, J.C.King,
An introduction to proving the correctness of programs,
Computing Surveys, vol.8#3, 76

[I4] C.A.R.Hoare,
An axiomatic basis for computer programming,
CACM, vol.12#10, Oct.69, pp.576-583

[I5] Z.Manna, S.Ness, J.Vuillemin,
Inductive methods for proving properties of programs,
CACM, vol.16#8, Aug.73, pp.491-502

[I6] C.A.R.Hoare, N.Wirth,
An axiomatic definition of the programming language PASCAL,
Acta Informatica, vol.2, 73, pp.335-355

[I7] W.A.Wulf, R.L.London, M.Shaw,
An introduction to the construction and verification of Alphard programs,
IEEE Transactions on Software Engineering, vol.SE-2#4, Dec.76, pp.253-265

[I8] A.S.Tanenbaum,
In defense of program testing or correctness proofs considered harmful,
SIGPLAN Notices, vol.11#5, May 76, pp.64-68

Testing a program means, running it to check that it meets its specifications. Debugging is the activity of locating and fixing errors in a program. Both activities are closely related in practical life, and the terms are often used as synonyms. To a large extent they are still considered more an art than an engineering procedure.

[J1, J2] provide overall surveys; [J1] contains an extensive bibliography on the subject. [J6] presents the current state-of-the-art of program test methods..

[J3, J4, J5] give guidelines resp. describe programs for the construction of test beds and test inputs. From the program and its specifications they deduce both typical test data, to exercise all alternative control paths, as well as extreme values that typically might escape proper treatment.

Testing by symbolic execution is a newer approach [J7], at least in its mechanical form. It can be seen as an extention and automation of the classical desk checking procedures, using symbolic values instead of actual (numerical) test values. The method is closely related to the program proving and verification approach, c.f. section I; the objections raised there are also valid here.

[J8, J9, L5] survey tools available for debugging. [J8] presents machine oriented facilities like core dumps and octal debug packages. [J9] treats high-level language oriented tools for execution profiling, source level dumps, etc.

The complete volume [A7] also contains worth-while articles on testing and debugging.

Recommended: J6, J1, J9

[J1] Ed: W.C.Hetzel,
Program test methods,
Prentice-Hall, 73

[J2] P.C.Poole,
Debugging and testing,
in: A3, pp.278-318

[J3] J.C.Huang,
An approach to program testing,
Computing Surveys, vol.7#3, Sept.73, pp.113-128

[J4] J.B.Goodenough, S.L.Gerhard,
Toward a theory of test data selection,
in: A7, pp.493-510
reprint: IEEE Transactions on Software Engineering,
vol.SE-1#2, June 77, pp.156-173

[J5] C.V.Ramamoorthy, S-B.F.Ho, W.T.Chen,
On the automated generation of program test data,
IEEE Transactions on Software Engineering, vol.SE-2#4,
Dec.76, pp.293-300

[J6] special section on testing,
IEEE Transactions on Software Engineering, vol.SE-2#3,
Sept.76, pp.194-231

[J7] J.C.King,
A new approach to program testing,
in: A7, pp.228-233

[J8] T.G.Evans, D.Darley,
Online debugging techniques, a survey,
FJCC 66, AFIPS vol.29, pp.37-50

[J9] E.Satterthwaite,
Debugging tools for high-level languages,
SOFTWARE Practice and Experience, vol.2#3, 72, pp.197-217

Some studies are available on the types, number and distribution of errors programmers make and the effort needed to correct them. In this respect the current section is related to section J on debugging and testing, but the emphasis here is not how to find and fix the bugs, but to investigate their origin and nature.

[K1, K2] are data collections on medium to large size assembly programs. They are helpful in classifying what kind of errors are actually made, e.g. wrong base registers, missing initializations, etc.. They also discuss the distribution of errors across modules, the way and effort needed to find them, etc.

[K3] describes a controlled experiment in programming language design using two small languages differing only in features thought to influence error frequencies like declarations and semicolon conventions. Students served as test persons. Though insufficient in several respects, the idea of using experiments of this kind in programming language design seems rather important.

[K4] analyses (subtle) errors in well-known articles about programming methodology and verification to show that these methods alone cannot guarantee against failure.

Recommended: K4

[K1] M.L.Shooman, M.I.Bolsky,
Types, distribution and test and correction times for programming errors,
in: A7, pp.347-357

[K2] A.Endres,
An analysis of errors and their causes in system programs,
in: A7, pp.327-336
reprint: IEEE Transactions on Software Engineering, vol.SE-1#2, June 75, pp.140-149

[K3] J.D.Gannon, J.J.Horning,
The impact of language design on the production of reliable software,
in: A7, pp.10-21
reprint: IEEE Transactions on Software Engineering, vol.SE-1#2, June 75, pp.179-191

[K4] S.L.Gerhard, L.Yelowitz,
Observations in applications of modern programming methodologies,
IEEE Transactions on Software Engineering, vol.SE-2#3, Sept.76, pp.195-207

An old saying tells: "Craftsmen are recognized by sharp tools." This also applies to the software engineer. In addition to the fundamental requirement of a programming language, treated in more detail in section M, he needs editors, filing systems, libraries, document formatting systems etc.

Surveys of such tools are given in [L1, L5]. A really excellent treatment of tools, their characteristics and everyday practical value for the programmer is given in [L2]. This book not only presents the tools themselves, but builds them up step by step, illustrating the program development process in a very instructive manner. Programs for text manipulation in general are treated in [L3] and for administration of releases and variants in [L4].

Recommended: L2

[L1] D.J.Reifer,
Automated aids for reliable software,
in: A7, pp.131-142

[L2] B.W.Kerningham, P.J.Plaugher,
Software tools,
Addison-Wesley, 76

[L3] A.v.Dam, D.E.Rice,
Online text editing: a survey,
Computing Surveys, vol.3#3, Sept.71, pp.93-114

[L4] M.J.Rochkind,
The source code control system,
IEEE Transactions on Software Engineering, vol.SE-1#4,
Dec.75, pp.364-370

[L5] B.W.Boehm, R.K.McClean, D.B.Urfrig,
Some experience with automated aids to the design of large scale reliable software,
in: A7, pp.105-113,
reprint: IEEE Transactions on Software Engineering,
vol.SE-1#1, March 75, pp.125-133

[L6] C.V.Ramamoorthy, S.F.Ho,
Testing large software with automated software evaluation systems,
in: A7, pp.382-394

The programming language is a fundamental tool for the software engineer, resp. programmer, and has been paid high attention in nearly all discussions on software engineering.

[M1, M2, M3, M4] form some kind of an encyclopedia on the subject. [M1, M3] are catalogs with short descriptions, whereas [M2] treats concepts of some commonly used programming languages. [M4] gives a nice account of the history of programming languages.

[M5, M6, M7] discuss language features for 'heavy duty programs'. [M7, M8] treat machine-near programming languages needed e.g. for writing operating systems or real-time systems.

[M8, M9, A9 (in part)] are proceedings of conferences dealing with programming language design. A (personal) review, resp. guideline for programming language design is given by Wirth in [M10], stressing simplicity and uniformity.

Uniformity and outer appearances of language constructs are further discussed in [M11].

Recommended: M4, M7, M10

[M1] J.E.Sammet,
Programming languages: history and fundamentals,
Prentice-Hall, 69

[M2] M.Elson,
Principles of programming languages,
SRA, 76

[M3] J.E.Samment,
Roster of programming languages for 74-75,
CACM, vol.19#12, Dec.76, pp.655-669

[M4] P.Wegner,
Programming languages - the first 25 years,
IEEE Transactions on computers, vol.C-25#12, Dec.76,
pp.1207-1225

[M5] G.Goos,
Language characteristics: programming language as a tool in writing system software,
in: A3, pp.47-69

[M6] G.Goos,
Systemprogrammiersprachen und strukturiertes Programmieren,
in: Hackl (ed.), Programming methodology,
LNCS 23, Springer 75

[M7] W.A.Wulf,
Issues in higher-level machine-oriented languages,
in: M8, pp.7-12

[M8] Ed: W.L.v.d.Poel, L.A.Maarsen,
Machine-oriented higher-level languages,
proc. of an IFIP-TC2 conference, Trondheim, Aug.73,
North-Holland, 74

[M9] Ed: D.B.Wortman,
proc. ACM conference on language design for reliable software,
SIGPLAN Notices, vol.12#3, March 77
some articles in: CACM, vol.20#8, Aug.77, pp.539-595

[M10] N.Wirth,
On the design of programming languages,
in: IFIP 74, North-Holland, 74, pp.386-393

[M11] C.M.Geschke, J.G.Mitchell,
On the problem of uniform references to data structures,
in: A7, pp.31-42
reprint: IEEE Transactions on Software Engineering,
vol.SE-1#2, June 75, pp.207-219

Software documentation is an all too often neglected activity. The material available mirrors this situation:

[N1] gives a short survey. [N2] points out the needs of the user. [N3] is just a forms cookbook. [N4, N5] present graphical rephrasing styles for code. [N6] proposes that the neglection of documentation be attacked by organizational procedures.

Recommended: N1

[N1] G.Goos,
Documentation,
in: A3, pp.385-394

[N2] N.Newman, T.Lang,
Documentation for computer users,
SOFTWARE Practice & Experience, vol.6#3, 76, pp.321-326

[N3] D.A.Walsh,
A guide for software documentation,
McGraw-Hill, 69

[N4] N.Chapin,
Flowcharting with the ANSI standard: a tutorial,
Computing Surveys, vol.2#2, 70, pp.119-146

[N5] I.Nassi, B.Shneiderman,
Flowchart techniques for structured programming,
SIGPLAN Notices, vol.8#8, 73

[N6] R.C.Fitzpatrick,
Making documentation painless,
Datamation, Aug.77, pp.62-68

According to [02] the term portability means the ease (or lack of it) with which a program may be moved from one machine to another. The term adaptability refers to the ease with which it can be changed to cater for different environments or functional changes. In this sense only portability is discussed in the papers cited. Adaptability is generally aimed at by using high-level languages (c.f. section M), even in machine-near programming [M8].

There are two schools:

- Abstract machine modelling, inventing 'virtual hardware' particular suited for the task at hand, and writing the software in this language. The virtual machine instructions must then be translated into the code of the real hardware at hand. Macro expansion is usually used for that task. Waite and Poole [02, 03, 04] pioneered this direction.

- The other approach to the problem is to use high-level programming languages. That these are not as portable as often claimed (especially FORTRAN) is nicely illustrated by [05]. Again two approaches:
  - Living in an imperfect world with language dialects, word sizes dependencies, etc., the NAG library [06], a large subroutine library for numerical computations, is preprocessed for each machine / compiler to produce appropriate variants.
  - The US Department of Defence (DOD) is in a much stronger position: They prepared a standard acceptance test for COBOL compilers [07]. By that de-facto standard imposed on the market COBOL compilers and programs processed by these compilers behave much more uniformly than e.g. FORTRAN systems.

A broad survey on all these approaches is given in [01].

Last but not least, a 'software porter' usually spends a high percentage of his time just converting character codes, fiddling strange tape formats and job contol languages. Some hints for dealing with these aspects of porting are given in [08].

Recommended: 01

[01] Ed: P.J.Brown,
Software Portability, An advanced course,
Cambridge University Press, 77

[02] P.C.Poole, W.M.Waite,
Portability and adaptability,
in: A3, pp.183-277

[03] M.C.Newey, P.C.Poole, W.M.Waite,
Abstract machine modelling to produce portable software - a review and evaluation,
SOFTWARE Practice & Experience, vol.2, 72, pp.107-136

[04] S.S.Coleman, P.C.Poole, W.M.Waite,
The mobile programming system, JANUS,
SOFTWARE Practice & Experience, vol.4#1, 74, pp.5-23

[05] M.A.Sabin,
Portability - some experience with FORTRAN,
SOFTWARE Practice & Experience, vol.6, 76, pp.393-396

[06] S.J.Hague, B.Ford,
Portability - prediction and correction,
SOFTWARE Practice & Experience, vol.6#1, 76, pp.61-69

[07] H.T.Hicks,
The Air Force COBOL validation system,
Datamation, Aug.69, pp.73-81

[08] W.M.Waite,
Hints on distributing portable software,
SOFTWARE Practice & Experience, vol.5#3, 75, pp.295-308

Striving for efficiency in an early phase of program development usually does much harm to the software produced. From Knuth [P1] we now know that in most cases 90% of the running time of a program is spend in less than 10% of its code. Time efficiency may therefore best achieved by first having a correct clean solution, then observing its dynamic behaviour (e.g. by tools as [P2, J8]) and reworking the critical parts. An even better solution is to leave this (partially) to an optimizing compiler [P4]. A good description of the process (and some advise on how to proceed) is given in [H15].

A completely different task is to measure (and maximize) the performance of a computer system as a whole. This is treated in the surveys [P5, P6].

Recommended: P1, P3

[P1] D.E.Knuth,
An empirical study of FORTRAN programs,
SOFTWARE Practice & Experience, vol.1#2, 71, pp.105-133

[P2] G.Lyon, R.B.Stillman,
Simple transforms for instrumenting FORTRAN decks,
SOFTWARE Practice & Experience, vol.5#4, 75, pp.347-358

[P3] D.B.Loveman,
Program improvement by source-to-source transformation,
JACM, vol.24#1, Jan.77, pp.121-145
1st version: 3rd ACM symposium on principles of programming languages, Jan.76

[P4] P.B.Schneck, E.Angel,
A FORTRAN to FORTRAN optimizing compiler,
Computer Journal, vol.16#4, pp.322-330

[P5] H.C.Lucas,
Performance evaluation and monitoring,
Computing Surveys, vol.3#3, Sept.71, pp.79-91

[P6] C.C.Gottlieb,
Performance measurement,
in: A3, pp.464-491

Software engineering is a difficult subject to teach at a university. Basic ingrediences of real-life projects can hardly be appreciated by the students. Consider for instance the sheer size and complexity of systems involved, the management problems, the cost and schedules competition in the software market. A more basic problem is that we do not yet have any satisfactory complete methodology, resp. professional image of a 'software engineer' to teach.

A basis for any new curriculum is the ACM curriculum [Q1, Q2].

[Q3, Q4, Q5] are attempts towards a software engineering curriculum, [Q4] giving the essentials, whereas [Q5] is a transscript of a workshop with positions and discussions from both university and industry.

[Q6] describes a term project which was very successful in illustrating the software engineering process.

Recommended: Q4, Q6

[Q1] ACM curriculum committee on computer science,
Curriculum 68 - recommendations for academic programs in computer science,
CACM, vol.11#3, March 68, pp.151-197

[Q2] ACM curriculum committee on computer science,
(Ed: R.H.Austing, B.H.Barnes, G.L.Engel)
A survey of the literature in computer science education since curriculum 68,
CACM, vol.20#1, Jan.77, pp.13-21

[Q3] P.Freeman, A.I.Wasserman, R.E.Fairley,
Essential elements of software engineering education,
in: A8, pp.116-122

[Q4] A.I.Wasserman, P.Freeman,
Software engineering concepts and computer science curricula
COMPUTER, June 77, pp.85-91

[Q5] Ed: A.I.Wasserman, P.Freeman,
Software engineering education,
proc. of an interface workshop,
Springer, 76

[Q6] J.J.Horning, D.B.Wortman,
Software hut: a computer program engineering project in the form of a game,
IEEE Transactions on software engineering, vol.SE-3#4,
July 77, pp.325-330

The following journals regularly contain articles on software engineering:

a) IEEE Transactions on Software Engineering

b) SOFTWARE - Practice and Experience

c) Communications of the ACM (CACM)

d) COMPUTER

e) Datamation

f) Computing Surveys

g) SIGPLAN Notices

h) Computing Reviews (indirectly, as it is a reference journal for short reviews of new publications)

Other bibliographies on the subject were prepared by students at the University of Toronto, Canada, Computer Systems Research Group:

An annotated bibliography on computer program engineering

- 3rd edition, CSRG-54, April 75
- 4th edition, CSRG-69, May 76
- 5th edition, CSRG-80, May 77