2002

Cutting Hacking: Breaking from Tradition

Rick Duley
Edith Cowan University

Stanislaw P. Maj
Edith Cowan University

10.1109/CSEE.2002.995214
This is an Author's Accepted Manuscript of: Duley, R. , & Maj, S. P. (2002). Cutting Hacking: Breaking from Tradition. Proceedings of 15th Conference on Software Engineering Education and Training. (pp. 224 - 233). USA. IEEE. Available here

Cutting Hacking : Breaking from Tradition

Rick Duley S P Maj
Edith Cowan University, Perth, Western Australia
{r.duley, p.maj} @cowan.edu.au

Abstract

Student memories of introductory programming units regularly centre on seemingly interminable cycles of coding and debugging and the WIMP[1] syndrome perpetuates this horror in the software development industry. Code construction is the one inescapable phase of the software development cycle yet educators seem unable to escape the mind-set which equates programming with syntax and semantics, condemning succeeding generations of students to this painful experience. Accumulated coding wisdom of more than four decades, now being codified in the SWEBOK, should smooth the process of learning to develop software if presented early in the student's career. Typically, however, it continues to be presented in advanced programming units (or in other, specialised, units) rather than as fundamental understanding. As a consequence, code hacking is the norm in undergraduate years despite being antithetical to the Software Engineering ethos.

This paper examines current practice and presents the view that what is needed is not a return to basics but rather an advance to basics — that syntax and semantics should be seen as a means of expression of formulated ideas and that student software engineers should be exposed immediately to the notion of code construction as the application of basic concepts rather than fluency in a language — thereby potentially bringing together programming practice and Software Engineering theory.

1. Introduction

Terminology

1. *'He, She' etc.* : used as generic terms except in obvious context
2. *'Course'* : refers to a three or four-year programme of study leading to the award of a degree
3. *'Unit'* : refers to a semester-long series of lectures, workshops, assignments etc. leading to a final assessment as part of a course

"All professions are conspiracies against the laity." So wrote George Bernard Shaw (The Doctor's Dilemma; *Act One*, 1906) and laymen confronted with the complexities of matters such as law or medicine – or an income tax return – ruefully agree. Engineering professionals acknowledge this, and even count on it, as Layton pointed out in 1972:

> *"…scientific knowledge possessed by the engineer is highly rational, but his professionalism derives from the mere possession of esoteric knowledge, not its specific content. Incomprehensibility to laymen, rather than rationality, is the foundation of professionalism."* [1]

Electronic computing, as a field, has not been immune to this accusation. Computer scientists and engineers, latterly including software engineers, have been seen by the uninitiated as acolytes of the arcane and esoteric. During the evolution of Software Engineering (SE), commentators have defined the concepts now being codified in the Software Engineering Body of Knowledge

[1] **W**hy **I**sn't **M**arvin **P**rogramming?

Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)
1093-0175/02 $17.00 © 2002 **IEEE**

(SWEBOK) and a vocabulary standardised by the ISO and the IEEE. These concepts and words epitomise the results of intensive deliberation directed towards the production of robust, reliable, maintainable software and many of them are, in the context of the computer world, old. We might expect that computing truths, once privileged to the few, are now freely available to the many – that experience and accumulated wisdom of (computing) generations is generously and widely disbursed to all who will hear. However, warning signals were being sounded a decade ago:

> *"If the critical truth is lacking in SE education, the obvious remedy will be to expose and transmit (or at least begin to transmit) this truth as early as possible in the education (of software engineers). The alternative to acquiring SE expertise during formal education is to learn it during professional practice — that is by learning from mistakes made on SE applications which are inherently complex, important and costly. Such a dangerous luxury our society cannot afford. Neither is it in the best interest of SE professionals."* [2]

Loyola espoused the theory that, given the boy at the age of seven he could return the man. Similarly, only by communicating this SE *truth* to programmers when they begin their training may we expect their return as software engineers.

2. Code Construction

> *"Programming is an essential skill that must be mastered by anyone studying Computer Science. Placing it early in the curriculum ensures that students have the necessary facility with programming when they enrol in intermediate and advanced courses."* [3]

Programming, however, is more than learning a language while being less than Software Development (SD). Still, central to SD is the process of *"writing a program"*, sometimes known as *constructing* a program. *'Code Construction'*, McConnell's term, fits well with the idea of Software Engineering, designating the process the indispensable phase of software development.

> *"The ideal software project goes through careful requirements analysis and architectural design before construction begins. The ideal project undergoes comprehensive, statistically controlled system testing after construction. Imperfect, real-world projects, however, often skip analysis and design to jump into construction. They drop testing because they have too many errors to fix and they've run out of time. But no matter how rushed or poorly planned a project is, you can't drop construction; it's where the rubber meets the road."* [4]

This critical operation is seen by the authors as the almost mechanical application of long established concepts and principles to the translation of problem solutions into an High Level Language and thence into a program of instructions for a computer — a perception which falls in line with the idea that the training given a traditional engineer permits people of ordinary talent to produce a certifiable product. Many of these concepts and principles are old in the context of digital computing, being defined and discussed decades ago:

(a) Sequence, Selection and Iteration (reference from 1966) : *"It is thus proved possible to completely describe a program by means of a formula containing the names of diagrams ? , ? and ? ."* [5]

(b) Subroutines and Abstraction (reference from 1972) : *"…the notion of the closed subroutine is still one of the key concepts in programming. …one of the greatest software inventions …it caters for the implementation of one of our basic patterns of abstraction."* [6]

(c) Cohesion and Coupling (reference from 1972) : *"The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system."*

[7]

(d) Complexity, Modularity and Top-Down Design (reference from 1979) : *"Complexity can be decreased …by breaking the problem into smaller and smaller pieces, so long as these pieces are relatively independent of each other."* [8]

Some made their appearance much later:

(e) Encapsulation and Information Hiding (reference from 1991) : *"…we have packaged a collection of data values together with the operations on those values. This packaging process is called Data Encapsulation. …the language does not allow the user …to be aware of, or to take advantage of, the internal implementation of this data structure… This masking of the internal implementation of a data type is called Information Hiding."* [9]

So independent are these concepts from implementation languages that it has been suggested that programming might be removed from CS1 altogether [10] yet IPUs continue to focus in their early stages on teaching language dependent syntax and semantics relegating conceptual understanding to later or other units. This, it seems to the authors, is akin to expecting a child to learn to communicate by studying a dictionary. In this analogy we advocate a natural process in which the child looks for words to express concepts, not that he merely tries out words in an effort to discover what they really mean. Common approaches to teaching programming, those starting with syntax and semantics, condemn the student to endless hours of experimentation, repeatedly coding, correcting and testing in an effort to get a statement in a prescribed language to *'work'* without first establishing what *'working'* really is.

> *'…concentrating on the mechanistic details of programming constructs often leaves students to figure out the essential character of programming through an ad hoc process of trial and error."* [3]

In our analogy, he has the words to use but doesn't know what he wants to talk about whereas in childhood, using his natural language, he knew what he wanted to talk about and struggled to find the words and grammar with which to express himself. This non-natural style of learning results in what is often euphemised as *'Exploratory Programming'* but which is, simply, *'Code Hacking'* and is anathematic to the ethos of Software Engineering.

Conceptual models presented in IPUs do not have to be comprehensive and detailed, indeed a danger may exist in presenting such models in that the students risk being over-powered by facts. Continuing the childhood analogy, a child does not need a meteorologist's model of a cloud to want to comment on the beauty of a sunset. Concept depth and detail may be introduced in parallel and later units. Tyro programmers tend to work with abstract, black-box models of computers and compilers anyway —programming concepts may, at first, be equally abstract.

2.1. Learning to Engineer Code

Learning an High Level Language (HLL) should not be difficult. In a western, developed nation a two-year-old typically learns an average of a new word every two hours for ten years — nearly 44,000 words — but there are only 69 reserved words in Ada95, most of which are already known to and contextually understood by a new university student. Certainly, natural languages allow ambiguity, synonymity and shades of meaning which are not permissible in computer languages, but problems with computer languages do not arise from their vocabulary but rather from its use — hardly a new concept, as Soloway wrote in back in 1986:

> *"New research with novice programmers… suggests that language constructs do not pose major stumbling blocks for novices learning to program. Rather, the real problems novices have lie in "putting the pieces together", composing and coordinating components of a program. …learning to program amounts to learning how to construct mechanisms*

and how to construct explanations. …In an introductory programming course, students should be taught what they really need to know about programming…" [11]

What, then, is programming? :

"It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex. It is the art of translating this design into an effective and accurate computer program." [12]

"Translating", in the view of the authors, is the keyword here, indicating the direction in which the process moves — from concept to language. Writing the actual code is seen as a late and minor step which can only be taken after Soloway's *'pieces'* are put together.

3. Teaching an SE IPU with Abstract Concepts

First impressions are critical. *"Hello World"* lacks impact for today's students and may be replaced with exercises more captivating to students. One example, often used by the authors in a first workshop, is this:

Some sentence such as *"Now is the time for all good men to come to the aid of the party."* is written on the board and a student is invited (coerced) to the board with instructions to *"think like a computer"* and to count the number of spaces in the sentence. Edith Cowan students quickly reply *"Fifteen."* Conversations such as the following ensue:

"How did you figure that out?"

"I counted them."

"But how did you count them?"

"I looked at the sentence and counted the spaces — like you asked."

"How did you know where the spaces were?"

"I looked at them."

"But you are a computer and a computer can't see — it wouldn't be able to see the difference between a blank character and a bottle of Coke. How do you — a computer — do it?"

"I look along the line and when I find a blank character I count it."

"Good. How do you know it is a blank character?"

"Well, I know what a blank character is and, if I see one I count it."

Ere long, with a succession of students, we arrive at a recipe for success (algorithm) such as:

```
1. take a piece of paper <memory> to keep the count on
2. repeat from the beginning until the end of the sentence
    look at a character
    compare it to a blank space
        if it is the same as a blank space add one to the count on the
            piece of paper
```

Now we have the concept — how do we express it so that a real computer can understand it and work on it? We need some words to express the concept — syntax and semantics related to the concept.

"…theoretical material is often presented before students have sufficient practical and scientific experience to follow what is presented and understand its significance." [13]

We need some memory, a repeating structure, a sentence to look at, a way to read the sentence, a way to know we have finished, a character, something to compare it to and a way to compare characters. All of this information can be elicited from students. In Ada95 we need the code shown in Code Sample 1.

Code Sample 1 : Count_Spaces

```
procedure Count_Spaces is
    Spaces      : Integer                   := 0;
    Sentence    : constant String(1 .. 65) :=
        "Now is the time for all good men"
        & " to come to the aid of the party.";
    Space       : constant Character       := ' ';
begin -- Count_Spaces
Count_Loop:
    for Letter in 1 .. 65 loop
        if Sentence(Letter) = Space
        then
            Spaces := Spaces + 1;
        end if; -- Sentence(Letter)
    end loop Count_Loop;
end Count_Spaces;
```

What is introduced? Basic program structure (procedure <> is … begin … end<>;), identifiers (what can be used, relevant wording), types (integer, string, character), type range (1 .. 64), operators (:=, &, ‘ etc.), initialisation at elaboration (Blanks : Integer :=0;), constants, iteration, assignment (:=), addressing (Sentence(Letter)), conditional statements (if <> then <> end if), the IDE, compilers.

SE concepts? Code that compiles — a regular refrain is *“Don’t hand me code with compiler errors.”* Code that doesn’t crash — another refrain is *“Don’t hand me programs which don’t work properly.”* Code layout and readability — *“Will you understand your own code in six months?”*. Verbose format (named parameters etc.) — *“Think about each and every step as you take it.”* Design first — *“Decide what you want to do before you start.”* Algorithm design — *“Write down the recipe.”* Meticulously systematic construction of code — *“Follow the established process.”* Familiarity with the Language Reference Manual — *“RTFM.”*

Even an IDE can be strange to many of our cross-disciplinary students, especially the mature-age ones, but it can be described in terms like *‘typewriter’*, and *‘page’*. Compilers? Just a *‘black box that does …’*. Language-dependent concepts? Always a problem, but here comparable to (in the authors’ view, industry best practice) a company providing an apprentice with a comprehensive tool-kit on day one. No apprentice is expected to know what all the tools do, many look at their kit mystified. Using them explains their functionality. In this exercise we say to majors, *“Don’t worry about that just yet, you’ll understand later in your course.”* Non-majors may remain satisfied with the higher-level abstraction.

Is this overloading students who might not even know how to turn a computer on? It has not proved to be so. In fact, the inevitable response when students run the program is that they want to do more. *“…but it didn’t do anything!”*, to which the reply is, *“Yes, it did — the computer just didn’t show you what it did. …That’s because you didn’t tell it to.”* This leads to a more complete exercise, introducing the idea of packages (Ada.Text_Io), which reads a line from the keyboard, counts the spaces and shows the result. Depending on the student group, one could then go on to the concept of files (Standard_Input), parameters (Get_Line(Item => <>, Last => <>)), and exceptions (CONSTRAINT_ERROR etc). All of this can readily treble the length of the program — which is, itself, a pertinent introduction to the complexity of IO operations — but the result is rather more satisfying than *‘Hello World’* and (in the authors’ experience) well within the

comprehension range of tyro programming students. This may appear to be *'dropping students in off the deep end'* but the authors' experience is that they swim.

Of major importance is that the syntax and semantics presented allow expression of a significant concept already established — tools to do a job rather than theoretical curiosities. This is the more natural method of learning a language. Instead of a student learning a word and searching for its meaning, she has reached a stage of having something she wishes to say (to make the computer do) and seeks the words to achieve that expression.

Another important outcome of this lesson is that students come to see computers as dumb machines which have to be told everything. Dumbness of the machines gives students a sense of power over them — destroying apprehension — and knowing that computers have to be told everything appears to relieve much of the initial frustration with them that students regularly experience. Instead of *"Why won't this darn thing do … ?"* we start to hear *"Oops, I forgot to tell it to…"*.

4. What We Actually Teach in IPUs

How do educators present programming as a subject? More specifically, how do educators present programming to novices. It is not uncommon for IPUs to be common units delivered to non-majors in computing and, for these people, an IPU may be the only programming, indeed the only computing, unit in their entire course. A conventional IPU can, however, be ineffective in communicating SE concepts. Even in matters as fundamental to the practice of programming as Sequence, Selection and Iteration, the change in student comprehension of the subject in an IPU at Edith Cowan University did not show as great a difference at the end of the unit as one might have hoped (Figure 1) [14]. Barely half of the students who had passed the unit were able to produce valid descriptions of the concepts. It has been suggested that they had dealt with these concepts under other names, (in point of fact, the authors believe that the problem was related to a concentration on syntax and semantics) but this brings us to another question, that of vocabulary. Computing is regularly strewn with new words, acronyms and catch-phrases and non-adherence to standardised vocabularies is a ready path to a lack of communication. Standard vocabularies, defined by both the ISO and the IEEE, are available but neither the educators nor the practitioners in the field seem to pay them more than scant attention.

> *"Computing professionals, too, have a standard vocabulary. …Yet dictionary makers and — far worse — most computing professionals have largely ignored the international standard vocabulary. …No computing textbooks seem to adhere to the standard."* [15]

Another aspect of the problem is that many computing faculty come from backgrounds in mathematics or electrical engineering. Actual CS graduates have been rare amongst faculty and few SE courses are sufficiently mature to supply their graduates into the field. Those educators committed to SE principles must carry a great responsibility here.

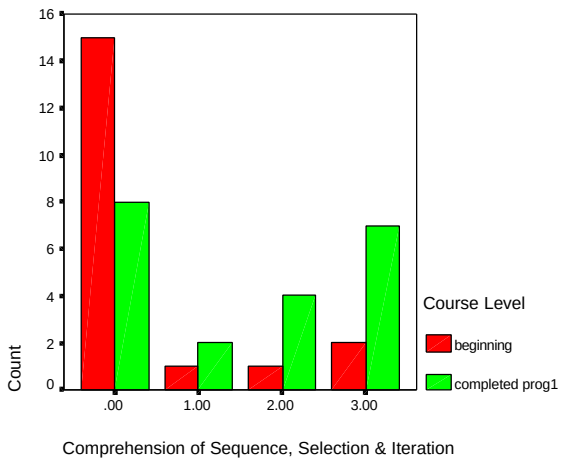

Figure 1 : Student Comprehension

4.1. Advertising the Concepts We Teach

Table 1 : Unit Outline Content

Unit Outlines (2000)	
Concept	**%**
Algorithms	41
Cohesion, Coupling	0
Control Structures	38
Data Structures	31
Design	41
Documentation	28
Encapsulation	9
Exception Handling	13
Identifiers	0
Information Hiding	9
Inheritance	9
Modularity	0
Object Oriented	44
Parameters	0
Polymorphism	3
Problem Decomposition	44
Procedural / Structured	16
Program Construction	28
Pseudocode	0
Recursion	6

Published unit outlines might be expected to show that the novice will receive a firm grounding in the concepts underpinning *'effective and accurate computer programs'*. Twenty-two IPUs from 11 Australian universities (of the 14 now offering SE degrees accredited by the Institute of Engineers, Australia, (IEAust)) including 11 units from SE degrees and 11 Computer Science (CS) degrees [16] — plus ten IPUs from universities not offering IEAust degrees — were reviewed by the authors. In Table 1 the first column refers to key words and phrases which might be referred to in an outline, column two to the percentage representation. As might be expected, approaches differed. Many referred to *Control Structures* while few actually named *Sequence*, *Selection* and *Iteration*. Three course outlines did not contain an identifiable, individual IPU and programming was dealt with over several interrelated units. Some other salient points are:

(a) Cohesion and Coupling, Data Validation, Identifiers, Modularity, Parameters and Pseudocode were not mentioned.
(b) Problem Solving and Design rated relatively well but SE themes like Data Validation and Documentation did not.
(c) Nearly half mention Object Orientation but less than 10% mention Inheritance.
(d) Generic programming concepts as listed above rated poorly.

Generally, appearances of these key words and phrases were much less frequent than the authors expected. Admittedly some unit outlines reviewed allowed very few lines for descriptive text and some avoided technical terminology almost completely but, overall, prospective unit content was poorly defined. What then, given such hazy definition of intent, do programming teachers actually say in their unit lectures?

4.2. Actual Content of our Lectures

Over 900 e-mail Requests for Information (RFIs) were transmitted to CS and SE educators in Australia, Europe, the UK, the USA and Asia asking for access to actual IPU lecture notes for the first three weeks of a semester. Three weeks was the chosen period because:

(a) three weeks represents 20-25% of a university semester, by the end of which period, in the authors' estimation, unit themes and emphases should be becoming clearly defined, and
(a) educators might, the authors felt, be reluctant to hand on a full set of notes to strangers while being more ready to let us study a sub-set of notes.

Replies originated from Scandinavia, the UK, the USA and Australia. Response was surprisingly low and while a minuscule sample cannot produce valid statistical data, there remain some pertinent commonalities which must be viewed in the light of their context.

4.2.1. General Context

(a) Individual course designers have individual priorities, grouped by CC2001 into six categories:
    1. programming-first (imperative-first)
    2. programming-first (objects-first)
    3. programming-first (functional-first)
    4. breadth-first

5. algorithms-first
6. hardware-first

with programming-first models remaining dominant.

(b) Choice of programming paradigm clearly affects the content of IPUs.

4.2.2. Common Factors

IPUs are unlikely to include reference to concepts such as dynamic memory allocation and addressing in the first weeks but a range of concepts more likely to appear and their appearance rates in the unit notes returned are shown in Table 2. Note:

(a) All included extensive reference to administrative details — most an entire lecture.

(b) Seven of the eight devoted most of the second two weeks to syntax and semantics.

(c) Concept references were low. Most common were

1. program construction or references to algorithm implementation as a technique which is part of the greater issue of programming (88%)
2. problem decomposition specifically (75%) or as a facet of problem solving (38%)
3. algorithms (63%)
4. identifiers and how we choose them (63%)
5. control structures (50%)
6. documentation (50%)
7. parameters (50%)

(d) References tended to be brief.

(e) No course referred to language independence in design or program writing technique.

Table 2 : Concept Reference

Concepts in Notes	
Concept	**%**
Algorithms	63
Cohesion & Coupling	13
Control Structures	50
Data Structures	13
Design	38
Documentation	50
Encapsulation	38
Exception Handling	13
Identifiers	63
Information Hiding	38
Inheritance	25
Modularisation	25
Object Oriented	50
Parameters	50
Polymorphism	0
Problem Decomposition	75
Procedural/Structured	25
Program Construction	88
Pseudocode	25
Recursion	0

5. Summary

Apparently, many IPU teachers accept that they do not communicate fundamental concepts and principles to beginner programmers, are hazy about what they intend to say in their lectures, and spend a lot of vital introductory time dealing with issues less crucial than the central theme of Code Construction. Faculty involved in IPU design are challenged to consider the following points.

5.1. Administrative Detail

Universities commonly lay great emphasis on administrative processes but the authors cannot help wondering if one whole lecture out of perhaps thirteen or fifteen needs to be devoted to what is often termed administrivia. These details are undoubtedly important, but need they take up valuable lecture time? Perhaps preparation of an on-line *'Administration Notices'* handout – and a requirement for an affidavit to the effect that it has been read (included in the document) to be printed, signed and handed in at lecture two – could free this influential week in which lasting first impressions are formed for more important matters.

5.2. Learning from Others

One of the great disappointments of this exercise was the low rate of response to the RFIs. It raises the question of the difficulty of learning from the efforts of others. It leaves an IPU designer

to operate on the basis of his own experience which may well be restricted to her student experience and the way she was taught the subject. Such a person becomes a variant of Bach's *'heroes'* [17] forced into the solitary re-invention of a wheel which, to the authors, appears to be regrettably out-of-round. Many faculty members involved in teaching programming are drawn from fields like mathematics or electronics engineering. This gives the *hero* a major problem when it comes to teaching IPUs to SE classes – she doesn't have that as student experience.

> *'Most of us teach what we were taught and teach it in the way we were taught it. Since nobody has yet graduated from a software engineering program …we will have to teach unfamiliar material…"* [18]

While IEEE-CS/ACM Computing Curricula include sample curricula, these are restricted to syllabus definition and leave a need for a library of Software Engineering IPUs as they are developed.

5.3. Breaking with Tradition

Teaching programming on a concepts-first basis might not be an original idea but the authors' reading has not included any publication on the theme over the past decade. This is despite the fact that well-known commentators stress the importance of teaching programming rather than teaching programming languages. Are SE educators really part of a *"Conspiracy against the Laity"*? Certainly not deliberately so in the experience of the authors, but we may be paying attention to business-related theories and processes to the detriment of the central issue of our discipline — teaching people how to create robust, reliable and efficient problem solutions and convert them into instructions for a rapid adding machine. Ongoing re-invention of an ineffective, legacy CS wheel is unlikely to provide a solution.

5.4. Student Immersion in Concepts

University students may be expected to be adults. Concepts presented in a *'black box'* form should hold no fear for them when they are assured that explanation and understanding lie further along in the course. That a new apprentice does not understand how a thread-cutting gearbox on a lathe works is no impediment to him learning how to thread a piece of metal. Understanding comes later. Successful use of the gearbox must come immediately.

It is the opinion of the authors that there are two evils in the presentation of concepts. Firstly that the teacher might shy away from them, effectively lowering the bar so that all students can step over it. Secondly that concepts are diligently explained before being applied, overloading students with detail and raising the bar so few can jump it. Abstract concepts give us a means to keep the bar sufficiently high to provide a useful challenge while being an achievable goal and the authors' experience is that students enjoy the triumph of jumping it.

7. Conclusions

Designers of IPUs within the *"programming first"* paradigm appear to be perpetuating the practice of concentration on syntax and semantics — possibly because that is the way they, themselves, were taught. This, to the authors, appears to be teaching a language the wrong way round. We hold that students should be encouraged (driven) to clearly establish first the idea of what they are trying to do, to think through a problem and to understand how to think a problem through. Then they can move to enunciating the solution in a manner in which a dumb machine can repeatedly and reliably calculate the answer(s). Then, and only then, need they seek the words in which to express their solution for the machine. This process more clearly reflects the natural process of learning a native language and may significantly reduce the trauma of *'learning to*

program'. Having a clear idea of the problem and a solution prior to writing code can also significantly reduce the time and effort regularly expended on the hacking cycle of *'code-and-debug'* which so often plagues undergraduate experience.

Reference List

[1] E.T. Layton. *The Revolt of the Engineers: Social responsibility and the American engineering profession*, London (UK): The John Hopkins University Press, 1986.

[2] A. Rosel, The Problem Solving Paradigm - A base for education of software engineers *Australian Computer Science Communications*, vol. 10, no. 1, pp. 269-279, Feb, 1990.

[3] IEEE-CS/ACM Joint Curriculum Task Force, Computing Curricula 2001 (Steelman Draft) ; Web Page - URL: http://computer.org/education/cc2001/steelman/cc2001/index.html (Accessed: Aug. 3, 2001)

[4] S. McConnell. *Code Complete*, Redmond, WA (USA): Microsoft Press, 1993.

[5] C. Bohm and G. Jacopini, Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules *Communications of the ACM*, vol. 9, no. 5, pp. 366-371, May, 1966.

[6] E.W. Dijkstra, The Humble Programmer *Communications of the ACM*, vol. 15, no. 10, pp. 859-866, Oct, 1972.

[7] D.L. Parnas, On the Criteria to be used in Decomposing Systems into Modules *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, Dec, 1972.

[8] E. Yourdon and L.L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Cliffs (NJ, USA): Prentice-Hall Inc., 1979.

[9] G.M. Schneider. *Concepts in Data Structures and Software Development: a text for a second course in computer science*, St Paul, MN (USA): West Publishing Co., 1991.

[10] R.L. Shackelford and R.J. LeBlanc, Depth First and Breadth First Models of Computing Curricula; Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education, pp. 6-10, ACM, New York, NY (USA).

[11] E. Soloway, Learning to Program = learning to construct mechanisms and explanations *Communications of the ACM*, vol. 29, no. 9, pp. 850-858, Sep, 1986.

[12] O. Astrachan, Education Goals and Priorities *ACM Computing Surveys*, vol. 28, no. 4es, 1996.

[13] C. Brown, H.J. Fell, V.K. Prolux, and R. Rasala, Instructional Frameworks: Toolkits and abstractions in introductory computer science; Proceedings of ACM Computer Science Conference, pp. 195-200, (1993). ACM, New York, NY (USA).

[14] R. Duley and S.P. Maj, Did We Really Teach That?: A glimpse of things students (don't) learn from traditional CS1; Proceedings of the 13th Conference on Software Engineering Education and Training, pp. 237-245, (2000). IEEE Computer Society, Los Alamitos, CA (USA).

[15] N. Holmes, The Great Term Robbery *IEEE Computer*, vol. 34, no. 5, pp. 94-96, May, 2001.

[16] R. Duley, D. Veal, and S.P. Maj, Educating Professional Software Engineers: Pathways and progress in the Australian experience; Proceedings of the 14th Conference on Software Engineering Education and Training, pp. 213-220, (2001). IEEE Computer Society, Los Alamitos, CA (USA).

[17] J. Bach, Enough about Process: What we need are heroes *IEEE Software*, pp. 96-98, Mar, 1995.

[18] D.L. Parnas, Software Engineering Programmes are not Computer Science Programmes *Annals of Software Engineering*, vol. 6, pp. 19-37, 1998.