Project Based Learning of Software

Richard Tabor GREENE

Professor of Knowledge and Creativity Management, School of Policy Studies, Sanda, Japan
https://www.youpublish.com/people/1201
http://www.scribd.com/people/view/310309
http://kwansei.academia.edu/RichardTaborGreene
email: richardtgreene@alum.mit.edu

Why programming languages cannot be taught—there is nothing there to teach
Why engineers fight marketers and vice versa making bad software
Why childhood is a bad anti-education idea—separating kids from the work contexts of ideas
Why you can learn all your days without becoming in any way educated
Why Flip defeated Sony, Hitachi, Toshiba, NEC, Panasonic in Videocamera Tech
How the internet is a mass media nemesis not a technology
The difference between coding and programming
Getting testesterone out of the economy, technology, computers, and software
The femininity of productivity and creativity

d

1 WHAT IS SOFTWARE AND HOW DO THE BEST AT IT DO IT

In 1962, just leaving middle school, I got a 3-month summer job writing a Fortran I program (executed on an IBM 1401) expanding the stress integrals used in designing bridges, binomially, keeping integer coefficients throughout, and integrating the resulting series. The next summer I got a similar job, this time using Lisp 1.5 newly mailed from MIT to a researcher near my house, to control a crystal growing machine. The summer before graduating from high school I got another 3-month programming job, this time using APL to scan photos of stars and map new stellar objects not on previous photos of the same portion of sky. When I entered MIT, in 1966, I took 6.251 the entry course to computer science there and horrified, realized the professor, John Donovan, wanted us to write a Cobol language compiler in PL-1. I did not consider Cobol and PL-1 to be proper parts of computer science, rather I thought of them as commercial shallow tools, so I dropped majoring in computer science formally and concentrated on playing with grad students at MIT's project MAC in the basements of Tech Square, where we had sometimes more than a dozen PDP computers and the rule was—never unplug anything because mistakes following cables often meant that what you unplugged stopped something vital to someone else. I wrote artificial intelligence programs in Lisp 1.5, Conniver, Sail, and some other rare languages for five years, all outside of formal courseware, but infinitely more rewarding because grad students would tear apart my code in instants, and my code had to work with what a dozen others were doing. I learned to program by being yelled at, basically, when my routines did not fit and do their assigned job when combined with routines by others. At MIT, my junior year, I wrote a blackboard architecture program in Lisp where words and sentence patterns in the database perused the blackboard for what a machine was trying to express or report to a human user, and automatically finished each sentence the user started to type. There was more to it than this. The grad students had a culture of their own we undergrads got embedded in. It was a culture of a few clear guidelines and principles:

1. coding is different than programming and programming is a lot better—when given a function to get software to do, the coder immediately starts banging out code on a computer while the programmer goes to Starbucks and writes models on napkins for days before writing shorter, more elegant, less errorful code
2. as Marve Minsky used to tell us—the most important piece of equipment for the good programmer is the napkin, because programming is pure thought and often done in lunch conversations where you bring a point that stymies you to the attention of uninvolved programmer friends who unstick your brain for you.
3. good programmers are between 80 and 150 times more productive than the best coders; good programs are between 1/80 and 1/150th as long as bad ones for achieving the same functionality
4. end users of programs are morons and nothing they say they want can be really trusted, you have to analyze their job a lot better than they do to be safe in executing a programming project for them with any hope for improving things—the actual end-user of any program is both the person using the application you write, and the managers and customers of that person in and outside his organization—they have to be well served by what your software application enables that employee to do
5. programming hardware and software evolves so rapidly that the natural language intent of each program part is the only thing that endures for 10 or more years
6. therefore, there are ten steps in creating a good program, and the 9th one is choosing which computer language to use, and the tenth one is translating the program, completed in step 8 in natural language expression form, into the computer language codes chosen to be used in step 9.
7. you represent the problem a dozen ways before choosing several to actually use in your program
8. you understand what operands need to be operated on in your problem domain and then you imagine what operators to apply to them to obtain the results you need
9. you invent a very abstract representation of the problem situation, used to specify very abstract operands and operators on those operands, so that in effect you invent a mini-language to express all of your problem/solution domain then use that language to specify your program
10. a good program will be two maybe three pages long whereas a coder, not a programmer, will write 50 or 100 pages of code for the same functions achieved
11. all good programs have the same two sections—a first section is code for a highly abstract language the programmer has invented that expresses the problem domain in a highly abstract way (consisting of operators applying to operands, both of which are highly abstract), and a second section wherein that the functions needed by the enduser get expressed in the grammar of that language; this means the language is capable of millions of functions beyond what the present enduser has requested which helps programmer look down on and feel generally superior to endusers and their short sighted requirements
12. the problem in collaborating with a group to write a program for a really large problem domain is specifying how everyone's program bits will interface with everyone else's—this was later solved by object oriented programming and later still by patterns (standard object sets for common programming contents and functions).
13. 90% of the program code for any software project is copied from code freely available on computer networks and good programmers end up writing program code only for the 10% of their application's

functions that are "interesting", that is, that require something original not already out there in someone else's application work—the 3rd or 4th step in programming is stealing code from elsewhere

14. Putting knowledge into programs so the ideas could recognize when they pertained and initiate actions was hindered by programs/computers not being raised for 20 years as humans are, so programs/computers lacked commonsense about how the basics of the world worked—this made for brittle tech systems that failed unexpectedly and fatally when encountering elementary aspects of reality no human would have to think about. For the long term we needed a global standard knowledge base on how every elementary aspect of the world worked, the every application could use, so systems would be less brittle.
15. Europeans wrongly thought programming was based somehow for some reason on human logic (Prolog type projects) while Americans wrongly thought programming could ignore logic and just take the shortest route to doing a needed function (C++ type projects)—it turned out American programs would beat European ones in the market at first because they were faster to produce though messy, but the European ones would win later because they were conceptually elegant and used deeper cleaner logic and maths, but still later the Americans would win eventually (CAD is exactly this story, early US wins superceded by CATIA's victory, now CATIA has been sloughed by IBM in favor of an improved US vendor).
16. Nearly all programmers are men and therefore computer hardware and software systematically everywhere all the time asks for and implements overly male versions of work processes that already, before the computer came along, were overly male—nearly all flaws, errors, and disasters related to computer technology come not from the technology but from the maleness of those doing, buying, using, and modifying it.
17. Creativity and productivity at work is nearly always achieved ONLY by feminizing systems of work.
18. Marketers, in top management, push software applications to be too subservient to moronic customer short sighted whims while engineers in top management, push software applications to be too oblivious to moronic customer short sighted whims.
19. Good sex is a necessary relief from looking at computer screens all day—without good sex the quality of programming work nose-dives dangerously; therefore, the best programmers tend to be a little more hansom than the average male in their own society.
20. Software people for some reason develop very fast minds and when they talk in public they speed up till they are talking so fast their tongues get cramps.
21. Good programmers are Star Trek and science fiction fans, and prefer Jean Luc Picard to that totalitarian Kirk captain in Star Trek first generation.

In later years after more interesting and challenging programming work, I develop better principles but none of those later principles contradicts in any significant way, the principles I list above. The principles above still apply as far as I have been able to determine. My later principles are contained in a book I wrote in 1990 by combining my homework from 2 master's and one Phd. Of courses from the University of Michigan—Global Quality, 1993, by McGraw Hill.

2 SOME REVOLUTIONS:

John Maeda, on of the inventors of the multi-media design computer language Processing at MIT's Media Lab, has just been named head of America's leading design college, the Rhode Island College of Design. Digital "materials" are the mainstream of broadcasting, narrowcasting, publishing, media, gaming, and design. It is perfectly apparent that newspapers, books, TV, collaboration, economies are changing economics, shape, and participation principles as they shift to the global web as their main platform and avenue of distribution and supply. This is a general digitization to all our lives and involvements. There are some quite general changes in how we imagine, feel, and think, going on in all fields, due to this encroachment of digital systems. Wolfram in a wonderful book A New Kind of Science, for example, suggests that quantum phenomena are a digital basis of reality (reality as one dimensional cellular automata, the simplest thinkable systems) peeking out from an analog smooth overlay from our human dailylife scale of living. Below I describe very briefly a few revolutions within this overall digitalization of life and systems in our time.

First, the discovery of machine computation (and actual building and use of it) spawned seeing computer-like and software-like behavior in the biologic and social systems around us. That, in turn, spawned the invention of new more biologic styles of machine computation and new more social styles of machine computation. Those two inventions, in turn, gave us the ability to spot still further computer-like and software-like aspects in biologic and social systems around us. This virtuous circle is still going on, gradually transforming the dominant metaphors, images, and research agendas in fields like geology, sociology, literature, and psychology. Everything is beginning to look, feel, and be treated as if it were software. In a previous publication I called this the "general empiric computation" model of the evolution of computer and software systems.

Second, in the mid-1980s total quality systems from Japan revolutionized all global businesses, switching everything from a vertical gaze (the boss' ass) to a horizontal gaze (the customer's needs). At the precise same time, technical developments were expanding lone computers from local area nets to wide area nets to nets across the internet. This coincidental development of a horizontal vision of how to define and run work and a horizontal net technical capability meant that total quality became the ideology, the theory, the framing viewpoint by which leaders bought, used, and modified technical systems now for several generations. Total quality was the theory of doing work that told us when and how to use web based computing to improve customer satisfaction, supplier cooperation, speed to market, and quality of outcome.

Third, the internet was a revolution in democracy hiding behind the appearance of being a technology. Venture software businesses in the US and other nations, were dominated by generations of students (average founders 39 years old!) and professors who hated central broadcast media and the way they vulgarized and monopolized performance, composing, and publishing during the 20th century. The internet was a way to democratize those functions, so we end up with 6 billion TV stations, publishers, performers, replacing rich drug-addict central elites in Tokyo, Paris, New York, Los Angeles, jails, and treatment facilities. People formed venture software business to change the world not to make money in nearly all cases—only inept MBAs formed ventures to make money and most of them failed as a result.

Search came to be the doorway and commercial footing of the web in the form of Google's victory over Microsoft and Yahoo. Search continues to increase in importance and cost as the web continues to grow without good indexing. The human brain—seeing and remembering everything in autism without meaning or pattern and seeing and remembering far less in normal healthy persons due to indexes that shut out most for the sake of purposeful percepts—is the opposite of the web now. The web is autistic! Self indexing and categorical regularization systems for the messy mass of what exists on the web and for entering new material onto the web are slowly emerging as attempts to enhance search are overwhelmed by the volumes and new kinds of web content emerging.

Fourth, there is the Palm defeating Sharp's long dominant Zaurus, there is the Flip defeating Japan's camcorder industry, there is the iPhone defeating global phone manufacturers. There is a consistent pattern of over-engineered industries upset by customer-oriented upstart firms and their upstart products. Technology industries are driven by engineers and engineering and drift, by natural force, away from customer needs till upstarts see an opportunity and embarrass everyone financially. Related to this is how companies that long thought of themselves as hardware firms, like Sony, are brought kicking and screaming into reality by new leaders and by competitors forcing them to realize that most of the value in the products they produce comes from software not hardware.

Fifth, huge industries are trying to sell technology things as the solutions to everything. Most of this is a lie. Software does not solve most problems. So paid people tell us that a software system will allow retired R&D people to help with problems in our firm—this is true, for a few months, but initial excitement decays and people newly united on a net get familiar then bored and participation drops to zero in a few years. Technology knows only how to further connect people and people need a rhythm of connection and isolation, creativity dies when connection is too great. We need, in truth, pulsed systems not connectedness systems.

Now, with the above tiny synopsis of my very personal, limited exposure to software, and perspectives on its forces and endpoints of evolution, I can turn to how I and other learn to do software well and how I and others learned to teach others to be effective at doing software well.

3 LEARNING AND TEACHING IN GENERAL

We can start with general dimensions of education and learning that apply regardless of the content involved. Then I will present a few additional points that come into play mostly when software and computer systems are involved.

1. The disaster for education that inventing childhood and public education were—about 100 years ago, give or take a few decades, the whole industrial world stripped kids from their parents and other adults and daily exposure to work and put them in child only containing buildings where they spend the first 2 decades of their lives memorizing verbal formulas about the world without making or doing anything with that "knowledge".

This is a cruel stupid way to create human beings—without context of exposure to actual adults and use of ideas.

2. Educating is the process of creating human beings, their minds. Factions, leaders, old rich elites, gurus, radical, religions—all like getting to kids early before they can judge for themselves and imbibing them in favored ideologies and lies about reality. Education systems are socially agreed on lying systems—for forming the kind of compliant, unquestioning, tax paying dummies elites find easy to manipulate and profit from.

If we want a scientific, safe, non-ideologic way to define education content, what procedure do we use? In previous publishings I have presented 48 capabilities of highly educated people, gotten by asking 315 eminent people in 63 professions, half US, half global, who the most educated-acting people they know are and what makes them call them "highly educated-acting", then asking 150 such people they nominated what constitutes highly educated-acting behavior in them and others.

Education differs from learning—you can learn all your days without for that reason in any way whatsoever becoming the least bit educated (a paraphrase of Hannah Arendt). Educating is about leading people out from when and where they were born and leading out from people a second spiritual rebrith to replace the local, biased, socially narrow birth in one language, nation, gender, and family we all start with.

3. Lectures are a publishing system not a teaching system and have been irrelevant to education purposes for many hundred years since the Chinese invention of movable type printing. That schools and colleges still depend on lectures is a measure of how little concerned they are with educating students. Similarly memorizing verbal content and formulas has nothing to do with education. The result of much memorization at national universities in Germany and Japan is what is commonly called "human fact vacuum cleaners", people who are well informed about anything without being able to synthesize

and think about anything they have information on. East Asian exam traditions from China perpetuate this memorization caused inability to use ideas and elites in East Asia are happy to prevent ordinary people from exercising judgment and thought—dumb populations are easier to be elite-r than than smart ones.

4. The entire industrial world lacks a safe, reliable, definition of what a good person is, what goodness itself is, and how to be a person for our current world and circumstances. We are unlikely to create people we all like being with and are safe being with till we reach some sort of consensus on what sort of thing goodness is, a good person is, and how to be such a person. Truth is, we lack, as entire societies, a goal, what to produce with our school systems, parenting, technical systems, and economies. The way the "best" colleges in the US created greedy self-deluding elites in MBA form that ruined their own wealth and the entire world economy in 2008 and 2009, demonstrates that the "best" people and colleges in the US do not have a clue as to what goodness is, what a good person is, and what sort of person it is safe to make and have and lead and be led by today. Our systems and societies are pointless and goal-less for the most part as this essay is being written.
5. Ideas for use, ideas make you more powerful, ideas are worth getting and using—these three propositions do not apply to the vast majority of education systems and their teachers and students. We design the first 20 years of all human lives (in industrial societies today) so that they master via thorough hourly and daily and yearly practice: sitting all day, getting fat (via sitting and eating suicidal foods from giant food industries), being passive, creating nothing, and conforming to a lot of petty rules. Such people make excellent employees, great people to kill in regular nationalism-caused or religion-caused wars, and compliant dummies for easy manipulation by lazy arrogant inherited wealth elites.
6. People are terrible at taking any idea and finding what part of their own life and experience it points to or relates to. People are terrible at taking any fact and seeing what procedure it changes or suggests doing. For the most part, people will not see what part of life an idea is talking about unless you specifically demonstrate what part of their particular life the idea applies to. This is called "grounding" work. People will not see what procedure needs changing and what change in it is needed that a particular fact implies unless you show them exactly what links that fact to past or new procedures.
7. People remember facts encountered in a procedural context and people remember facts they have to ask others about or explain to others in team based settings. Individual study and work is very ineffective compared to team and project combined as a context for learning and remembering and recalling and using and extending facts.

I could go on and on, and have elaborated the above points and many more in several of the books I have published (see Getting Real about Creativity in Business, Are You Educated? Japan, Are You Educated? 48 Capabilities at https://www.youpublish.com/people/1201). The above five points, however, suffice for my purpose in this essay on project based software education.

4　APPLYING THE ABOVE ABOVE TO LEARNING AND TEACHING SOFTWARE

All the above contents—the 20 principles of programming from my early MIT days, the revolutions ushered into our lives by digital systems, the points on learning and educating in general from immediately above—tell us quite specific things about how to conduct learning and teaching of software. There are many other frameworks I could refer to and include here in this essay but the above 3 frameworks are enough to make my main points on Project Based Software Education. Before applying the 3 above frameworks to software teaching and learning, I have five case examples of project based software education to present.

5 FIVE PROJECT- BASED SOFTWARE EDUCATION SYSTEMS AS GUIDES, INSPIRATIONS, AND EXAMPLES

Mitchel Resnick, for many years, a part of MIT's Media Lab, ran an experiment in public schools near Boston, in getting young kids to use a parallel processing population of interacting intelligent agents computer language called Star Logo to get on-screen and later on-floor mechanical turtles to do useful and interesting things. His goal was to indirectly instill in kids the assumption that populations of agents could coordinate and get complex results achieved without any leaders, one-man-up-front, and without any commands and commanders. Negotiation, consensus, provisional teams and solving events sufficed to allow diverse, seething populations to achieve complex outcomes. He wanted to counter the assumptions in society that we need male white over paid commanders and leaders to coordinate us and get us to get things done. He wanted to instill a more democratic commonsense about how coordination happens among ants, termites, whales, and every creature except mankind.

Malone, also at MIT, developed coordination theory, showing how the web lowered coordination costs across all functions in our world, dissolving both hierarchies, and the need for them, and replacing such vertical monkey-hierarchy systems with negotiating/collaborating horizontal webs of volunteers and participants.

Richard Stallman and a host of others spawned the open software movement which evolved into the open courseware movement and is still evolving into other open systems movements. The idea is revolutionary—the web makes it possible for anyone to communicate with anyone else in the world all the time, for free (nearly). This changes everything. In ***this*** context, it is evil to lock up the world's best teaching and knowledge in places where you need US$50,000+ a year plus to get access to it. Great universities manifestly do not emphasize teaching and get funding for and by research they do, yet they take tuition money meant by students and parents for teaching and use it for research instead. Open educuation and open courseware keep research monye for research and education money for education, thereby making the teaching and knowledge of the world's best universities available to any motivated person in the world for free (though a degree costs great money but even that border is being obviated by a new committee of professors from 20 universities who are creating an institution to administer exams for any opencourseware course in the world and award degrees for rational sets of such passed exams).

I myself at the University of Chicago created a software factory course, outlined for Aizu University in Japan as a chapter in my book Global Quality (reference above), wherein students used total quality methods to specify functions needed by a work process so its outcomes would better please the customers of the process, then used total quality program-composition events to compose code to do those functions. In two 12 week one-quarter courses of homework, classwork, and exam, we delivered working application, error free, that accomplished precisely what customers of a client work process wanted to improve their level of satisfaction with that process' outcome. The courses had a specification memory-check-enforcement team, an interface team, a data/knowledge-base team, an operators/operands-language-tools team, a code-scarfing/stealing team, and a customer communication-control- "kill bad ideas" team. Three monthly "builds" united code by the separate teams. Bi-weekly "builds" united code by individuals in each team. Yearly "builds" united code from two courses in series.

Yamaha's Tenori-On is a square board of buttons where you compose music the same way you compose computer software. It has 16 or so layers so depending on what layer you indicate, the meaning of the buttons changes, but in every case, the layout, spatially, of the buttons you choose to push, is your software and a repeating loop of flashing lights where buttons are pushed, executes what your spatial layout of pushed buttons "means". The layers and their respective pushed button patterns are the software program and the repeating flashing light loop of execution is the computer that software "runs" on. Making the composing of music much the same as composing a software program is the idea of the Tenori-Ons two inventors. They report on the web they had this intuition that composing music and software are the same, but no device, till theirs, made doing both the same.

At Kwansei Gakuin University I have two lecture classes in which students as homework and classwork have to turn social and personal situations into an abstract hardware "machine" that repeatedly executes the successive lines of code in some program people or society generates. Seeing the hardware nature of your own family's dynamics and the kind of procedural programs running on that emotional and social hardware, greatly extends the computation idea for students. Changing the software programs of self and family, while emotional work and therefore not easy, readily comes to mind once students realize the unexamined, happenstance, and often suboptimal nature of the programs that now run on their family-as-hardware computer or self-as-hardware computer.

I invented some years ago the Social Art Automaton. This is a way of composing artworks the same way software programs are composed, which is much the same way music gets composed on the Tenori-On. This Social Art Automaton idea is well enough developed that I present the main ideas in it in the section below.

6 GREENE'S SOCIAL ART AUTOMATON—COMPOSING ART THE WAY PROGRAMS ARE COMPOSED

A social art automaton consists of hardware upon which we run software. There are many forms of this and many ways to do the hardware and software functions involved. Here I present only a brief summary, leaving out most ideas for the sake of that special clarity that belongs to brevity.

6.1 Hardware:

- the general—issues instructions which are part of the software

one person, a committee, the audience (voting etc.), or the artists themselves by play this role

- teams—execute instructions

teams of 4 or 6 artists each can move to parts of the canvass or move along paths across it; fixed arrays of one team assigned to each section of a canvass can be used; mixed teams fixed arrays where team members from diverse teams are scattered among each other yet stay in fixed positions also can be used

- canvass—is what is modified, painted, shaped, sensed or otherwise turned into art by the application of somethings

flat, vertically sticking up shapes, intersecting walls, hangings, and random shaped areas are some of the types that can be used

- interval signal—this is a bell or whistle or chant which tells everyone when a new instruction is issued to all teams by the general—repeated short intervals of 2 or 5 minutes are generally used for 2 or 3 hours of art creation time

when chants are used, the painting or doing of the art can involve the teams or artists chanting or singing or otherwise turning their doings into performance

- rule/operation book—this is instruction summaries for the general, team members, or whole teams

operations per artist, per team, per section of canvass etc. can be devised

- toolings—the operations that teams apply to canvass parts to which they are assigned may involve tools of various sorts

for painting automatons: color, texture (screen, cloth, brush), substance (paint, glitter, sands, jewelry etc.), applier (spray, brush, drip, shoe covers walked, bodies painted and rolled etc.)

6.2 Software:

- the general's instructions—these tell artist units what operations to apply to what canvass section or painted/sculpted thing, of self or adjacent others, with what tooling and direction and location and shape

team X do operation X to section X in color X using texture X applied by X

- locations—paths, sections, regions, borders

numbered in order sections, numbered in order sets of sections (regions), numbered paths of the same shape, numbered paths of diverse shapes, preset paths, random paths, random paths from artist/team interaction rules

- instruction/operation types

absolute operations—apply X to Y with Z: relative operations—connect X and Y from adjacent sections, fix worst X in top adjacent section, copy largest curved shape in rightmost adjacent section, interpolate between largest colored section on left and on right, etc.

- team types—there are various types

art operation execution teams, edit/fix teams, make art teams, relate art object teams. Etc.

- cadence of call outs—the rhythm with which instructions are called out to all artist units

chants assigned per artist/team, chants assigned per operation type, chants assigned per object that operations apply to.

The operation of any social art automaton is a cadence rhythm called out at intervals of which the general issues instructions to all artists/teams. Artists/teams can change location as one type of such instruction.

7 APPLYING ALL THE ABOVE TO SPECIFYING HOW TO "TEACH" PROGRAMMING AND SOFTWARE

I am going to describe the conclusion I draw from the above, not anything scientific. This is a pre-science study for hypotheses to be tested rather than a study based on already established hypotheses. My candidate hypotheses below will be stark, somewhat exaggerated, to be clear, and at time, funny. My goal is not to be pompously impressively academic, but rather, true.

- No one ever learned to program well from any course or class. No one has designed a course or class that transforms people unable to program into people able to program well. In order to create such a class you have to so stretch what a class is, that use of the word "class" becomes entirely distortive and misleading.

People learn to program by enjoyably working in a community who share a common mission and who exchange various sorts of personal, financial, food, humor, social support, technical, training support among each other, while building something significant out of software.

- Colleges by emphasizing computer languages completely miss teaching programming and end up teaching coding. All of the bad programmers in industry come from such college experiences.

You teach programming by challenging people to write programs in natural languages after inventing minimal languages that suffice to allow all needed functions to be achieved by combining words grammatically in the invented language—you code the language and write in that language the functions needed.

- Colleges are filled with hackers who instantly go to coding when given problems, proud of their speed and decisiveness—such program end up being errorful and overly long and complicated, poorly thought through

You teach that programming requires napkins not computers—it is thought and design, not coding

- Programmers, unlike coders, invent abstract languages and write sentences to execute functions, this is why they achieve in1/80th code functions that coders achieve

You teach the abstraction processes of programming, not coding.

- All programs have managers paying for them and customers paying for them, both of whom have to be satisfied with the program—colleges by omitting total quality customer satisfaction techniques insure their "programmer" students write bad programs

You specify what functions to deliver so that the customers of the process the functions are used in get more satisfied with the outcomes of the process—a total quality spec of all software applications

- Nothing technical, in the hardware or software, of current systems will endure for ten years as effective content.

So colleges need to teach the natural language form of the program is all that endure through time, not the code or hardware it is implemented on or for

- You create the entire computer program without knowing what programming language you will use to implement it on some machine

You teach that the natural language form of the program is the program, and the code version is an implementation of the more general and abstract natural language form

- The first steps in programming involve representing aspects of the problem to be solved abstractly in several diverse ways; the problem a program is to solve can never be solved by the program alone but always involves a social life around the technical systems upon which they depend and forms of that, changes in that social surround have to be specified with each program as "the solution". There is no code only solution to anything in this world.

You teach inventing a program and inventing changes in the social life of info around the program and its users and that combination is the solution.

- Programs intermediately take the form of lists of operands to be operated on and lists of operations to perform on them—you invent a new minimal language to express all of those items, the sentences of that language doing those operands and operators is your program.

Invent a language to express the operators needed to apply to the operands needed and express those operands and operators in that language as what your program is

- Good programs for a function typically are 1/80 to 1/200th the size of bad ones: in my experience my best programmer employees could do 18 months work of my best coders in 4 hours on one afternoon of good thinking.

The abstraction factor makes programmers vastly more productive than coders

- Large groups cooperating on the writing of one program need, mostly, standard interfaces each module can depend on for communicating to other modules (furnished by object libraries and pattern libraries)

Mastery of standard libraries is a primary skill but it takes too long for any one college course to do.

- Actual programmers as their first step do a total quality spec of what the application must do, and as their second step try to find proven, un-errorful code, on the web, available for free, that they can borrow, steal, or modify—90% of actual programs are not written but scarfed from work by previous others.

Teaching students to find code is more important than teaching them to write code.

- The brittleness of programs, lacking commonsense from lacking being raised, means a social support system, a social life of info, surrounds each program and makes it work—software projects always deliver both a software entity and a social entity around it to make it work

Designing reward, organization, power, political, mindset changes and getting them implemented is half of all programming work.

- The first versions of any program are usually US, made clean and logical and clear by later Europeans, then made more automated by later US re-additions

A program's development always takes 20 years on two or more continents

- Both the software itself, those doing it, those using it, are all excessively male and you have to continually counter the excess maleness there

Students have to learn to spot excess maleness in themselves, in others, and in systems and tools

- Any system that improves any system makes that system more feminine

Students have to learn how to feminize themselves and systems of others

- There is a continual fight between marketers in management and engineers in management—they hate each other—good systems have to blend them both

If marketers win, technical capability suffers; if engineers win, customer use and satisfaction suffer—neither must be allowed to win—preventing such victory is a major skill of any programmer

- Programmers not good at sex, sports, or something else physical will burn up and die, ruining systems along the way

The mentality bias of software people is self destructive and programmers must be created to avoid it

- Software people must alternate between talking to other software people and talking outside that community

Programmers too alone with other programmers lose ability to function and specify and build systems that work.

8 REVOLUTIONS

- the software idea is us (our DNA), is quantum gravity (info theories of the universe), is sociality (cultures as software running on civilizations)

everything will be understood and managed as software for a while into the future

- total quality is the theory of software specification and use
- the internet is not a technology but the invention of defeats for central broadcast media and their preventing the rest of us from performing and composing
- the internet needs search because it is autistic—tools for regularizing inputs to the web will gradually erode the power and need for search
- engineers dominate systems till marketers amaze with systems customers like better, then engineers encroach and pollute again till later marketers repeat the cycle
- we need software the helps isolate people, and we need pulsed systems, with a rhythm of more connection and more isolation, not merely more and more and more connection without end or balance

9 TEACHING AND LEARNING IN GENERAL

- education in general teaches not making outputs of value—software courses the homework of which is sold to actual customers for money would solve this
- software applications that do not educate people or that are used by uneducated people are useless pieces of junk
- lectures are a publishing not teaching systems and need to be replaced by problems worked on in class with info provided by the professor wandering among groups who in procedural contexts need factual info on something
- technical people and engineers get used all their lives by richer, smarter, more socially aware people from humanities, social science, and professoinal programs—technical education that lacks robust social skills is suicidal

- societies educate to make people passive consumers—so software to assist existing societies often are evil in intent and effect—technical programmers unable to handle or discern this, are evil too
- people do not ground ideas—you have to show them how ideas relate to their lives; people do not transform facts into procedures—you have to show what procedures are implied by a fact

10 EXAMPLE PROJECTS

- software without central male command and control styles and values and assumptions serves to liberate imaginations of kids learning it—collaboration systems are going to replace command ones
- low cost coordination make markets replace hierarchies—collaboration systems are going to replace command ones
- restricting good education to rich elites by high tuition is evil and the best programmers in the world will in the future be created outside famous university courses
- selling homework to customers in a software factory two semester course thrills students and makes them work hard, because failure is real, not just a grade
- composing music via shaping software repeatedly executed typifies trends for all things to be created in software programming style
- hardware and software, applied as ideas to social systems of students' lives and families and careers help students grasp software ideas deeply and want to use them
- composing other arts, besides music, via social automata is a typical example of how software ideas are permeating all functions of life and society.