

Good Programmers are Not Lazy

Cort Dougan

Director of Engineering, Finite State Machine Labs

cort@fsmllabs.com

<http://www2.fsmllabs.com/~cort/>

Abstract

In this article I provide a practical analysis of programming practices and education. The phrase “Good programmers are lazy” is very often offered to students as a guide to good programming. This statement is very wrong and reflects some of the problems with attitudes in computer science today.

1 Lazy Programmers are Bad Programmers

In this paper I present some basic ideas for how computer science education and attitudes are flawed and should be reformed. This is not an irrelevantly academic treatise that waxes philosophical about the art of programming. Instead, it provides critical analysis of trends and attitudes that must be stopped, what has likely caused these trends, as well as offering real-world advice on what to do to end these problems.

“Good programmers are lazy” is a phrase

that is repeated and adhered to as much as “go to use considered harmful” is. There is rarely understanding behind either statement, though. Few people who quote Dijkstra’s paper[6] on the use of goto have even read it (even worse, few actually thought about it). Instead, they’ve heard the phrase and take the literal meaning of it as a guide instead of studying the actual cautions presented in the paper.

I’ve met and worked with lazy programmers and every one of them have been very bad programmers. A better statement would be “Good programmers find efficient, maintainable and thoroughly thought out solutions to problems”. This admonishment to be lazy is usually made by some well meaning and ignorant instructors while students are studying in college. I was given the same advice when I was a student but I have learned better since then. Good programmers are hard working, methodical, determined and driven – not lazy. The path of least resistance is for dairy cows, not for intelligent engineers and programmers.

2 Programming as Engineering

It is useful to compare programming practices with engineering practices. The field of engineering has drawn upon hundreds of years of experience in refining its methods. These methods are conservative, study failures in method as well as implementation in great detail and establish standards and certifications. Computer science can find very useful ideas in engineering.

As a means of comparison, what would you think if you were told we put an airplane together and it worked when we taxied around the airport so we know it's ready to fly now. It's a bit of an exaggeration but it's closer to the methods employed and taught by computer programmers than those taught and used by aeronautical engineers.

What I suggest is that computer scientists should be taught that one must “prove” an algorithm or program works in theory, “prove” it works in simulation and then “prove” it works as a real program.

Students must also be taught that even then “proof” is not real – even in the world of computer science or mathematics. Algorithms have been “proven” to be right and then found to be actually wrong[9]. There are errors in mathematical models, there are errors in understanding of the problem being solved and there are errors in implementation. People are sometimes wrong, designs are sometimes wrong and nothing one can do will ever completely eliminate that possibility. We must strive to remove as much chance

of error as possible in our methods. We cannot accept that any set of tests or proofs will be a guarantee. Testing can demonstrate the presence of bugs but not their absence[5].

2.1 Computer Science vs. Programming

The terms “computer science” and “programming” are often used interchangeably. There is a not so subtle difference between the two, though. The scientific study of computing, computer science, does involve programming but the opposite is not true. Programming is more closely related to an engineering discipline that benefits from the advances in its related scientific field.

However, A professional in either field should be well versed in its counterpart. This article is meant to be advice more to programmers simply because the aim here is to improve the methodology of programming.

2.2 Certification

Computer science needs certification of programmers and other professionals in the field. Certification programs such as Microsoft's MCSE, RedHat's RHCE and FSM-Labs RTLCE (along with the many others) are good beginnings but are not a complete answer. Programs that begin at universities to ensure that programmers have a good foundation for working in the industry and have mastered fundamentals are key to the success and improvement in the field. After-education training in specific packages

or niche areas are not the solution - they are a short-term workaround.

The ACM[3] has formally rejected the idea of certification and formal methods of professional assessment being used in computer science. A useful quote from their executive summary is “Such licensing practices would give false assurances of competence even if the body of knowledge were mature” [3]. This indicates to me that the ACM believes that all computer science professionals are incompetent and there is no method for making certain otherwise. This is definitely a terrible answer from a body that should uphold the reputation and ideals of computer science. An excellent counter-argument[11] in parody form was written by David Parnas that points out the flaws in the ACM Council position.

Other engineering fields have professional exams and certifications that must be maintained to be certain that practitioners stay current on the state of the art and have mastered a set of skills considered to be essential. Programming needs a similar system. We should all push for genuine certification of programmers – it will add to the quality of the work in this field. Parnas[12] writes about computer science graduates finding themselves in engineering positions and being unable to perform genuine computer program engineering tasks due to lack of training. He also lays out a program of computer program engineering that focuses on fundamentals and classical engineering principles.

We must avoid half-educated programmers from bad undergraduate programs designed to turn out java and HTML programmers

who end up working on critical systems. Even worse, there are programmers who have only had a high-school education and just seem to “be good at it”. These scenarios and how they really do play out in real life have detailed at length[1].

It has been claimed that certification of engineers who design buildings that people will actually occupy need to be certified but programmers do not. I wouldn’t live in a building designed by an uncertified engineer any more than I would want to live in a building that was designed with software written by an incompetent programmer. Programming is becoming more and more important to society just as engineering became critical with the industrial revolution. I believe that programming must become a rigorous field and more structured just as engineering has.

Computer science is guilty of hubris in believing it that it can escape the realities that other fields have had to face. It is also guilty of having shunned the solutions other fields have used because computer science is believed to be somehow different. Computer science is slightly different in a sense because it is a marriage of engineering, science and mathematics. Each of these fields has its own set of guidelines and rules that it can bring to computer science, though. Early computer science progressed quickly because computer scientists at the time were actually electrical engineers and mathematicians who had the benefit of the rigors of training those fields provide. We’ve lost that recently as computer science has become viewed as something different. It may be partly due to the ease of use many computers now advertise along with

the more and more simple programming languages. While the programming languages may be simple to use, correct programming is not any simpler because of them. The languages just hide the complexities. We cannot ignore the lessons that have been learned by the fields that have given rise computer science - we must add to them.

2.3 Programmers are Engineers – not Artists

In the field of computer science I have run across accomplished persons who have explained in detail how they have revolutionized the field through a mathematical insight that is generally taught in high-school. I have also seen programmers quit jobs because they were tired of the formal approach to problem solving that larger companies took. Fed up with being required to document their ideas before writing code they move on to less structured start-ups full of fresh out of high-school hackers. This prima donna attitude is a detriment to their ability and continued learning. It should have been corrected (or not instilled in the first place) in their early computer science education.

Moving to a more traditional approach to the science will be met with resistance because change is not easy. Moving from the easy-going field that computer science is now to a more defined and monitored discipline won't be well received. It is important and I believe will eventually be necessary.

3 Open-Source Hasn't Helped

I speak from having been maintainer and primary author of Linux on the PowerPC from its beginning for 8 years. I'm not an outsider to the open-source community and I've seen and been a part of the inner workings of its flagship project, Linux, for some time. Open-source hasn't helped the problem of poorly designed and engineered systems at all, it has aggravated it. The open-source community has contributed some valuable things to computer science but it has also led to many problems.

The wave of claims that Open Source will make your whites whiter and your servers faster are myths promulgated by well intentioned, but overly enthusiastic, supporters of open-source.

While the media has hyped the revolutionary development process of open-source, knowledgeable and highly-regarded professionals such as Rob Pike have derided the decline and desperate situation of operating system research and new developments[13]. The bursting of the bubble for business in the open-source and dot-coms area has been followed by a not widely recognized technological bust. Design and good engineering have been sacrificed for releasing code faster and more often. This "release faster and more often" ideology is a trademark of the open-source movement. Again, with Linux, the only innovative part of it may be the way it is developed. Linus Torvalds, leader of the Linux effort and ostensibly its chief de-

signer, admonishes programmers to avoid design of Linux and instead it should be allowed to “evolve”[18]. Avoiding design is definitely a revolutionary way to develop an operating system.

3.1 Testing

Successful Open-Source projects generally have a large user-base. A surprisingly large percentage of users are also willing to download and try out very new versions of the software. In part, making users a piece of the development group is the allure of Open-Source. This means serious errors tend to be found and reported by users reasonably quickly. Traditional software development tries to release only once testing is complete and so testing is considered part of the development process.

This removal of testing as a part of the development process weakens the engineering being done in Open-Source projects. If a programmer (or group of programmers) knows that the next release does not have to be functional because there is no penalty then there is no motivation to test. Users have become accustomed to non-working Open-Source software and so developers are not pressed to provide it with every release.

There is also no tightly connected group developing the software so there is no single plan. Several aspects of the system will be on different schedules and it’s unlike that every part will be complete or stable at the same time. Scheduling the development of software projects is difficult but it’s something that must be done to pro-

duce software. General open-source programmers seem to believe that software is so different from other disciplines that it cannot be planned[8]. Many outlets for younger programmers (such as slashdot.org[14] and developer.com[4]) grabbed onto this and took the ideas presented as far more reaching than they actually are. In short, the claim has become that programmers are artists that cannot be rushed. The attitude that software is different from other human endeavors and cannot be scheduled is the exact prima donna attitude that must be squashed. This attitude may simply be due to an ignorance of other fields and how they are similar to computer science but that is a problem in itself. Study in some of the more mature disciplines would help us greatly.

3.2 Apprenticeship is Gone - Both a Good and Bad Thing

Open-Source allows programmers to get to work on “real” problems and projects right away when it would not have been possible before. Before some of these projects, recently graduated programmers would have had to join a development group and serve an apprenticeship while learning more of the craft of programming. I benefited from the advantage that open-source gives people but I also fear the severe dangers of it and have seen some of the results of its downside.

The lack of apprenticeship is a problem because one doesn’t serve in the trenches and watch those who went before do the work.

Valuable lessons are learned in those situations that can't be taught during 4 years in a classroom. Rather than passing on knowledge and experience from "generation to generation" of programmers we're leaving people to their own. They must make all the same mistakes, form the same mistaken ideas and eventually after years of working they may actually find themselves at the level of skill that the previous generation peaked at. Rather than knowledge accumulating over time, knowledge and skill must be learned from the beginning with each new group of programmers. This slows progress of the science immensely!

3.3 Innovation

There is little innovation in open source. Microsoft claimed[2] that there was no innovation in open-source and the open-source community response[21] was to announce that the Microsoft invented .NET was going to be implemented by the open-source community. That not only shows lack of innovation but an entire lack of self-analysis or even an understanding of the problem.

The flagship of open-source, Linux, is not innovative either. It's an implementation of a 30 year old system that has changed little in that time. It's already lost much of its chance of being innovative that it once had because it's now in use by industry and must grow and be shaped to be a useful tool in business rather than a research and experimentation tool for students (where it's better used).

The popular open-source license, GPL, stifles innovation. Innovation is not easy. It's

not something that is cheap or quick. People, groups and companies must invest large amounts of time and money in something that may fail entirely. It's a risky exercise that few undertake. One must have a reason to risk, then. What reasons are there to risk all this? Money and fame are the main reasons. There is not much room for either when people start giving their software and innovations away.

Lately, Linux and open-source centered companies have taken to either becoming charities[15] or repeatedly appealing for donations in order to continue business[16][17]. Software is not a charity, businesses are not altruistic entities and innovation and solid engineering cannot come from desperate organizations. The open-source philosophy and GPL-centric licenses cannot sustain the science of computing which depends on large investments of time, effort and thought that must be rewarded to continue.

Again, programmers can learn from traditional engineers. If one looks at the field that is most closely tied to programming, microprocessor manufacturing, one sees an excellent model. Hardware companies profit greatly from their innovations and that, in turn, fuels more innovation. This has set in motion a great cascade of improvements in hardware every year that is unrivaled. This is the situation that we want to establish in the software industry. Stronger protection of intellectual property (in the form of software patents), better engineering practices and a more rigorous approach to design and testing can achieve this.

Innovation is necessary for the long term

survival of academic computer science and the computer software industry. It must be regained.

4 Ludicrous Ideas

Extreme Programming[19] is not a recent idea in programming. Its name implies snowboards, jolt cola and bungee cords – which is not far off. Its core idea is to put a group of programmers around a single computer and begin writing code without forethought or design in long sessions. Jolt cola and marathon programming sessions were a regular part, and a formative part, of my undergraduate programming education but it taught me that it is a recipe for bad programming. Extreme Programming is not a new idea, it’s just a new name. It’s traditionally referred to as “bad programming practice”.

It also advocates integrating code with other modules very often – sometimes several times per day. This couples distinct modules so closely that modularity is often lost. It also leads to, or is a result of, poorly defined communication between modules. This tight coupling of the development of different modules also makes parallel development and testing difficult[10].

This is typically the kind of programming that is found in programming contests. Hastily written programs that perform a given task with little testing and nearly no planning.

4.1 Languages are Not the Answer

The trend in programming languages it toward very simple languages with iterators, abstract data types that have been defined already and many complex operations such as mutual exclusion and memory allocation handled for the programmer. Languages with garbage collection and sophisticated data types hide complexities from programmers. These languages are useful for teaching programming but are dangerous for production use.

There are programmers who don’t understand the idea of mutual exclusion or how to synchronize processes and threads safely. They have no concept of the intricacies of locking because it has always been handled by the language.

Management of memory is another area where some programmers lack much needed skill.

The use of the word “hack” seems to fit into a culture of unskilled programmers. Avoid making quick and untested changes without design. Even simple changes that are obvious can lead to errors.

5 What to Do

What should one do, then? Here I provide some practical examples of what to do in every day programming and design. These ideas are not obscure and are obvious but are often neglected. Actually *applying* these ideas is what has helped me in my own pro-

programming and with those who I have worked with.

It's necessary to make programmers un-lazy. For a culture that has involved finding shortcuts and being rewarded for being "clever" this is not so easy. Many disasters in engineering have involved shortcuts that turned out to have made the critical difference in some very large system. This is an obvious disaster in mechanical design (bridges, automobiles and the like) but very rarely are they as exposed and studied in programming. Exposing errors, studying them and finding where they were introduced and what problem in the development process allowed them (or caused them) will reduce the likelihood that more will appear again.

Programmers often have fragile egos so it's difficult to point out failures and mistakes without a confrontation. Removing the problem from the programmer and placing it with the system is often useful.

5.1 Don't Allow Yourself to Become Lazy

Most programmers are actually not lazy by choice, but through lack of attention. Maintaining a list of tasks that must be completed as a checklist can keep people from sliding into laziness. A design document that includes an ordered list of milestones and acceptance criteria for completion of each milestone makes sure that everything gets done.

5.2 Design then Write

Don't write first! Design first. This is obvious but is very often neglected. An oft quoted adage in programming is "Things should be designed top down, except for the first time". This suggests that the best way to understand a problem and design a solution is to begin to attempt to write a program.

The best way to understand a problem is to write a thorough design for a solution but not the actual program. There's no need to write entire programs in pseudo code but a good collection of pieces include input, output and error conditions.

This idea has been documented very well elsewhere - in nearly every programming textbook, in fact. The key is to actually execute those methods.

5.3 Write an Example

Defining tests and criteria for passing those tests before writing a program is an excellent way to begin. This is a task that is vague and hard to define. So much so that the result is often frustrating and useless rather than productive.

In my experience it has been much easier to manage projects that begin with team members writing a man page, article or how-to for the system being constructed. It concisely describes what the end result should be. It is also a very useful exercise for thinking about the system in terms of the end - of the goal. Even an outline of these documents is useful. Examples can very easily become tests. If the system does not match the documentation -

or the examples - then the system is flawed.

These ideas are very useful, very simple and are also very rarely taught to programmers.

5.4 Don't Defer Until Later

First try it, if it fails then design. This isn't wise and something that is obviously going to fail but it's what programmers usually do. Every piece of code seems simple and quick to write until getting into it. Just like Christmas morning, putting together the kids toys before reading the directions, it often fails.

Things are even worse if it doesn't fail the first time because you're less likely to go through and do it right (design your code) if it already works. "If it ain't broke, don't fix it" makes sense if things aren't broken. However, it is broken, you just don't recognize it!

This most often happens when one writes a piece of prototype code or proof-of-concept. This is an ok practice of that is part of the design rather than becoming the design. Trying to graft more features and extend a piece of prototype code to produce a final product is a terrible chore and usually results in a very unmaintainable piece of code.

5.5 Test in Isolation

Breaking a problem or project down into discrete modules, even when how each module will function is not known, is a common design strategy that is often avoided. The design of large industrial systems is done this way so that components that are complete

can be tested with a simulation of the components that are incomplete. This also allows simulation of the communication between components in the system. Students are often told of this method, practice a few contrived exercises to use this method but rarely are they required to use it. It certainly doesn't last once they move into industry.

5.6 Be Methodical

You can't be completely methodical or remember everything even if you didn't have to worry about life outside programming. It's a very hard thing to do.

Write scripts to do things for you. Especially when doing performance tests. The benefits of automation in testing are well known[7]. However, it's rarely done because programmers are used to not testing methodically and waiting for the tests to complete.

"It doesn't work the same way it did yesterday" is a way of saying "I haven't been methodical enough". Either you changed something and don't remember or your test didn't encapsulate all the variables in the experiment. It's possible to review a test that is implemented with scripts but it is not possible to remember something you may have forgotten already. You cannot be certain.

You are not nearly as methodical as a computer and you shouldn't try to be. The computer is a very valuable tool that does not get bored doing the same thing over and over - but humans do. Avoid the temptation to take shortcuts (even unconsciously) by avoiding boredom in the first place. Let the scripts do the boring tasks for you so that you are

sure that you can reproduce your actions *exactly* in 5 minutes and in 5 months.

5.7 Lab Notes and Scientific Method

Many scientific disciplines have refined methodology for achieving reproducibility, avoiding common mistakes and making peer review easy and useful. Lab notes, lab reports, rigorous experiments and analysis are trappings of sciences that are notably absent in computer science and programming.

Above all these, guiding scientists in their work, is the scientific method. Although scientific method is very well known to most people, it might be useful to review it quickly:

- 1. Observation and description of a phenomenon or group of phenomena.
- 2. Formulation of a hypothesis to explain the phenomena. In computer science the hypothesis often takes the form of a causal mechanism or a mathematical relation.
- 3. Use of the hypothesis to predict the existence of other phenomena or to predict quantitatively the results of new observations.
- 4. Performance of experimental tests of the predictions by several independent experimenters and properly performed experiments.

These methods obviously apply to computer science. When debugging a program

or trying to find the cause of a given program's behavior. Many programmers apply these methods but do so incorrectly because the application lacks scientific rigor. Many computer scientists should read [20] for an review of applying scientific method correctly.

5.8 Test What You Know Already Works

The whole idea of testing is that you may have missed something, your understanding may be incorrect or the whole thing just plain doesn't work. Thinking that you can skip a test or two because there's no conceivable way your changes could have affected that test circumvents the whole idea of testing. You test because you're not perfect and can't see where a failure could occur. Everything should be tested as a black box because that's how it will likely be used. You must test as though you didn't write the code.

5.9 Fire Software Testers

Taichi Ohno[10] offers advice to businesses that want to improve efficiency and quality – fire your quality assurance engineers. His suggestion does not show a lack of interest in assuring quality but rather in improving it. His argument is that once a person working on an assembly line knows that work will be tested by a quality assurance engineer motivation to check over ones own work is gone. The same is true of programmers.

While getting rid of software testers is probably not a good solution it does point the

way to the solution. Programmers who know that software testers will catch errors are more likely to be lax and not be as thoughtful or design as carefully. Blame for a bug that slips through is moved away from tests and back to the programmers and program managers - where it is more useful. Blame being placed on testers may make better testers but don't we want better programmers (and thus better programs)? Automated regression tests, specification tests and build tests are valuable tools that have their place but they cannot be a crutch.

Programmers often sneer at testers as less skilled programmers. This isn't necessarily the case. In fact, programmers should become more skilled testers. Testing is a part of programming and should not be separated out as two independent tasks performed by different people. The loop has to be closed in order for a programmer to recognize systemic mistakes or mistakes in thinking quickly to correct the problem. A tester reporting an error through a bug-reporting system days or weeks later is not fast enough to help.

Programmers will likely not like being compared to assembly line workers but I see that there is great similarity between our profession and theirs. It is very useful to take lessons from what has worked for assembly lines and factories. Many of the ideas in this paper and the realization that computer science can gain a lot from factories came from readings of books by Taichi Ohno about improving quality and resource use in Toyota.

5.10 Know What You're Doing at all Times

A useful exercise I used for someone I worked with once was to stop him every 10 minutes and ask him what he was trying to do. He would often be at a loss and was unable to explain what he was trying to solve even after thinking for 5 minutes. It's easy to get lost when tracking down a problem since it often means studying different things to understand the problem as a whole. This can easily lead to aimless wandering through different pieces of code following ever-changing lines of thought. A constant reminder to yourself of what you are trying to do and how doing exactly what you are doing now is going to achieve that goal is valuable.

5.11 Have a Start, Have a Finish

Many programming projects become chaotic because of a lack of milestones and definite completion points.

It is necessary for a programmer to decide how a project will end before beginning. This obvious but it is rarely actually done. Large scale beginnings, endings and milestones along the way are common and are a part of any effort but it's not a common practice for individual programmers.

6 Reality

Programmers can't do a perfect job all the time. In fact, a programmer shows skill in

knowing when it's possible to get away with taking shortcuts when faced with real life constraints. A programmer should be at least aware of what is **not** being done right in order to be careful.

In an attempt to avoid the “Ready, Fire, Aim” problem in programming it's also important to avoid the “Aim, aim, aim, aim...” situation[1] which is nearly as common. There are times when it's necessary to actually begin writing code rather than continue design because of deadlines, time constraints and allocation of resource problems (such as available manpower!). There are ways to do this in an intelligent and planned way, though.

References

- [1] Dan Burke and Alan Morrison. *Business @ the Speed of Stupid*. Perseus, 2001.
- [2] CNet. <http://news.cnet.com/investor/news/newsitem/0-9900-1028-4825719-RHAT.html>.
- [3] ACM Council. A summary of the acm position on software engineering as a licensed engineering profession, July 2000.
- [4] developer.com. http://softwaredev.earthweb.com/msnet/microsoft/article/0,,10720_949921,00.html, March 2002.
- [5] Edsger Dijkstra. The humble programmer - 1972 turing award lecture. *Comm. ACM*, 15:859–866, October 1972.
- [6] Edsger W. Dijkstra. Go to statement considered harmful. In *Communications of the ACM*, volume 11, pages 147–148, March 1968.
- [7] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley Longman, Inc., Reading, MA, 1999.
- [8] J. P. Lewis. Large limits to software estimation. *ACM Software Engineering Notes*, 26:54–59, 2002.
- [9] John P. Lehoczky Lui Sha, Ragnathan Rajkumar. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.
- [10] Taichi Ohno. *Toyota Production System*. Productivity Press, 1988.
- [11] David Lorge Parnas. Two positions on licensing. In *4th International Conference on Requirements Engineering (ICRE'00)*, page 154.
- [12] David Lorge Parnas. Education for computing professionals. *Computer*, 23:17–22, 1990.
- [13] Rob Pike. Systems software research research is irrelevant.
- [14] Slashdot. <http://developers.slashdot.org/article.pl?sid=01/11/05/1410251&mode=thread&tid=156>, March 2002.
- [15] Slashdot. <http://mail.gnome.org/archives/foundation-announce/>

2002-December/msg00004.html, December 2002.

- [16] MandrakeSoft team. <http://www.mandrakelinux.com/en/future.php3>, December 2002.
- [17] MandrakeSoft team. <http://www.linux-mandrake.com/en/mdkfuture.php3>, March 2002.
- [18] Linus Torvalds. <http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.0/0004.html>.
- [19] Don Wells. The rules and practices of extreme programming. <http://www.extremeprogramming.org/rules.html>.
- [20] Edgar Bright Wilson. *An Introduction to Scientific Research*. McGraw-Hill, 1952.
- [21] ZDNet. <http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2820553,00.html>.