How To Be a Programmer: A Short, Comprehensive, and Personal Summary

Robert L. Read <read@hire.com>

December 16, 2002

Dedicated to the Programmers of Hire.com

Contents

1 Introduction

To be a good programmer is difficult and noble. The hardest part of making real a collective vision of a software project is dealing with one's coworkers and customers. Writing computer programs is important and takes great intelligence and skill. But it is really child's play compared to everything else that a good programmer must do to make a software system that succeeds for both the customer and myriad colleagues for whom she is partially responsible. Computer programming is taught in courses. The excellent books The Pragmatic Programmer[8], Code Complete[3], Rapid Development[2], and Extreme Programming Explained[4] all teach computer programming and the larger issues of being a good programmer. The essays of Paul Graham[6] and Eric Raymond[7] should certainly be read before reading this article. This essay differs from those excellent works by emphasizing social problems and comprehensively summarizing the entire set of necessary skills as I see them.

This is very subjective. This essay is therefore doomed to be personal and somewhat opinionated. I confine myself to problems that a programmer is very likely to have to face in her work. Many of these problems and their solutions are so general to the human condition that I will probably seem preachy. I hope in spite of this that this book will be useful. I have attempted to summarize as concisely as possible those things that I wish someone had explained to me when I was twenty-one.

In this book the term *boss* to refer to whomever gives you projects to do. I use the words *business*, *company*, and *tribe*, synonymously except that business connotes moneymaking, company connotes the modern workplace and tribe is generally the people you share loyalty with.

Welcome to the tribe.

Part I
Beginner

2 Personal Skills

2.1 Learn to Debug

Debugging is the cornerstone of being a programmer. The first meaning of the word to remove errors, but the meaning that matters is *to see into the execution of a program by examining it.* A programmer that cannot debug effectively is blind.

Idealists may think that design, or analysis, or complexity theory, or whatnot, is more fundamental, but they are not working programmers. The working programmer does not live in an ideal world. Even if she is perfect, she is surrounded by and must interact with code written by major software companies, organizations like GNU, and her colleagues. Most of this code is imperfect and imperfectly documented. Without the ability to gain visibility into the execution of this code the slightest bump will throw her permanently. Often this visibility can only be gained by experimentation, that is, debugging.

Debugging is about the running of programs, not programs themselves. If you buy something from a major software company, you usually don't get to see the program. But there will still arise places where the code does not conform to the documentation (crashing your entire machine is a common and spectacular example), or where the documentation is mute. More commonly, the programmer creates an error, examines the code she wrote and has no clue how the error can be occurring. Inevitably, this means some assumption she is making is not quite correct, or some condition arises that she did not anticipate. Sometimes the magic trick of staring into the source code works. When it doesn't, she must debug.

To get visibility into the execution of a program one must be able to execute the code and observe something about it. Sometimes this is visible, like what is being displayed on a screen, or the delay between two events. In many other cases, it involves things that are not meant to be visible, like the state of some variables inside the code, or which lines of code are actually being executed, or whether certain assertions hold across a complicated data structure. These hidden things must be revealed.

The common ways of looking into the innards of an executing program can be categorized as:

- using a debugging tool,
- printlining — making a temporary modification to the program, typically adding lines that print information out, and
- logging — creating a permanent window into the programs execution in the form of a log.

Debugging tools are wonderful when they work, but the other two methods are even more important. Debugging tools often lag behind language development, so at any point in time they may not be available. Because the debugging tool may change the way the program executes in subtle ways it may not be practical in all cases. Finally, there are some kinds of debugging, such as checking an assertion against a large data structure, that require writing code no matter what. It is good to know how to use debugging tools when they are stable, but it is critical to be able to employ the other two methods.

Some beginners have a subconscious fear of debugging in the sense of modifying and executing code. This is understandable—it is a little like exploratory surgery. But beginners have to learn to poke at the code to make it jump. They have to learn to experiment on it, and that nothing they can do to it will make it any worse. If you are a teacher or mentor for these timid people, please help them get over this fear by gently showing them how to do it and holding their hand if you have to. We lose a lot of good programmers at the delicate beginning to that fear.

2.2 How to Debug by Splitting the Problem Space

Debugging is fun, because it begins with a mystery. You think it should do something, but instead it does something else. It is not always quite so simple—any examples I can give will be contrived compared to what sometimes happens in practice. Debugging requires creativity and ingenuity. If there is a single key to debugging is to use the *divide and conquer* technique on the mystery.

Suppose for example we created a program that should do about 10 things in a sequence. When we run it and it crashes. We didn't program it to do that, so now we have a mystery—"It crashes." We can see that it did the first #7 by just looking at the output. The last three would not be visible from the output, so now our mystery is smaller "It crashed on thing #8, thing #9, or thing #10". Can we design an experiment to see which thing it crashed on? Sure. We can use a debugger or we can add printline statements (or the equivalent in whatever language you are working in) after #8 and #9. Then we run it again, our mystery will be smaller, such as "It crashed on thing #9." I find that bearing in mind exactly what the mystery is at any point in time helps keep one focused. When several people are working together under pressure on a problem it can get pretty confusing.

The key to divide and conquer as a debugging technique is the same as it is for algorithm design. As long as you do a good job splitting the mystery in the middle, you won't have to split it too many times, and you will be debugging quickly. But what is the middle of a mystery? There is where true creativity and experience comes in. To a true beginner, the space of possible errors looks like the lines in the source code. She doesn't have the vision she will later develop to see the other dimensions of the program, such as the space of executed lines, the data structure, the memory management, the interaction with foreign code, the code that is risky and the code that is simple. These other dimensions let an experienced programmer form an imperfect but very useful mental model of all the things that can go wrong. Having that mental model is what helps one find the middle of the mystery effectively.

Once you have evenly subdivided the space of all that can go wrong, you must try to decide in which space the error lies. In the simple case where the mystery is "Is the single line that makes it crash executed before, or after this line is executed?" you can just observe which lines get executed and you are done. In other cases your subdivision of the mystery will be more like "Either there is a pointer in that graph that points to the wrong node, or my the algorithm that adds up the variables in that graph doesn't work." In that case you may have to write a small program to check that the pointers in the graph are all correct in order to decide which part of the subdivided mystery can be eliminated.

2.3 How to Debug Using a Log

Logging is the practice of writing a system so that it produces a sequence of informative records called a log. The practice of printlining is just producing a simple and usually temporary log. Generally logging means a permanent and possibly configurable log, that offers these advantages:

- The log may provide useful information about bugs that are hard to reproduce (such as those that occur in the production environment but cannot be reproduced in the test environment.)
- The log can provide statistical and performance information.
- If configurable, the log may allow general information to be captured to debug a specific problem that would be difficult to reinstate for every problem if it were done on a temporary basis.

The amount of information output into the log is always a compromise between information and brevity. Too much information makes the log expensive and hard to use, too little information and it may not contain the information you need. Making the amount of information output configurable is very useful in this regard. Typically, each record in the log will identify its position in the source code, the thread that executed it if that is an issue, the precise time of execution, and commonly an additional useful piece of information, such as the value of some variable, the amount of free memory, the number of data objects, etc. These log statements are sprinkled throughout the source code but in particular at major functionality points, and around risky code. Each statement can be assigned a level and will only output a record if the system is currently configured to output that level. One should try to anticipate where problems might occur and design the log statements to try to address those problems. The need to measure the performance of particular subsystems is usually easy to anticipate and quite appropriate for a log as it allows the performance data to be collected in every environment.

If you have a permanent log, printlining can now be done in terms of the log records, and some of the debugging statements will probably be permanently added to the logging system.

2.4 How to Understand Performance Problems

Understanding how to learn about the performance of a running system is unavoidable for the same reason that debugging is. Even if the code you write considers performance perfectly, the demands placed upon it will change, the hardware it uses will change, and the software systems it interfaces with will be hidden and perhaps surprising in terms of their performance. However, in practice performance problems are a little different and a little easier than debugging in general.

Suppose that you consider a system or a subsystem to be too slow. Before you try to make it faster, you must build a mental model of why it is slow. To do this you have to know where the time or other resource is really being spent. A profiling tool or a good log is very useful for this. There is a famous dictum that 90% of the time will be spent in 10% of the code. I would add to that however the importance input/output expense (I/O) to performance issues. Often most of the time is spent in I/O in one way or another. Finding the expensive I/O and the expensive 10% of the code is a good first step.

There are many dimensions to the performance of a computer system, and many resources consumed. The first resource to measure is wall-clock time, the total time that passes for the computation. However, this may not be the whole picture. Sometimes something that takes a little longer but doesn't burn up so many processor seconds will be much better in computing environment you actually have to deal with. Similarly, memory, network bandwidth, database or other server accesses may in the end be far more expensive than processor seconds.

Contention for shared resources that are synchronized can cause deadlock and starvation. Deadlock is the inability to proceed because of improper synchronization or resource demands. Starvation is the failure to schedule a component properly. If it can be anticipated, it is best to have a way of measuring this contention from the start of your project. Even if this contention does not occur, it is very helpful to be able to assert that with confidence. Logging wall-clock time is particularly valuable because it can inform about unpredictable circumstance that arise in situations where other profiling is impractical.

2.5 How to Fix Performance Problems

Most software projects can be made ten to one-hundred times faster than they are at the code complete date (the earliest date that all code is completely functional) with relatively little effort. Under time-to-market-pressure, it is both wise and effective to choose a solution that gets the job done simply and quickly, but less efficiently than some other solution. However, performance is a part of usability, and often it must eventually be considered more carefully.

The key to improving the performance of a very complicated system is to analyze it well enough to find the rough spots. These places where most of the resources are consumed are then directly improved. There is not much sense in optimizing a function that accounts for only 1% of the computation time. You should do a performance analysis first to find out where the time is really going, and decide what to improve from there. As a rule of thumb you should think carefully before doing anything unless you think it is going to make the system or a significant part of it at least twice as fast. There is usually a way to do this. Work on anything less than doubling the performance only after you've thought carefully about the big wins. One reason to do this is to consider the test and quality assurance effort that will have to be applied to your change. Each change brings a test burden with it, so it is much better to have a few big changes.

After you've made a two-fold improvement in something, you need to at least rethink and perhaps reanalyze to discover the next-most-rough spot in the system, and attack that to get another two-fold improvement.

Often, the rough spots in performance will be an example of counting cows by counting legs and dividing by four, instead of counting heads. For example, I've made errors such as failing to provide a relational database system with a proper index on a column I look up a lot, which probably made it twenty times slower at least. Other examples include doing unnecessary I/O in inner loops, leaving in debugging statements that are no longer needed, unnecessary memory allocation, and in particular inexpert use of libraries and other subsystems that are often poorly documented with respect to performance. This kind of improvement is sometimes called low-hanging fruit, meaning that it can be easily picked to provide some benefit.

What do you do when you start to run out of low-hanging fruit? Well, you can reach higher, or chop the tree down. You can continue making small improvements or you can seriously redesign a system or a subsystem. (This is a great opportunity to use your skills as a good programmer, not only in the new design but also in convincing your boss that this is a good idea.) Before you argue for the redesign of a subsystem, you should ask yourself if you can make it five to ten time better.

2.6 How to Optimize Loops

Sometimes one encounter loops, or recursive functions, that take a long time and are rough spots in your product. You can probably make the loop a little faster, but spend a few minutes considering if there is a way to remove it entirely. Would a different algorithm do it? Could you compute that while computing something else? If you can't find away around it or you choose not to, then you can optimize the loop. This is simple; move stuff out. In the end, this will require not only ingenuity but also an understanding of the expense of each kind of statement and expression. But here are some suggestions:

- Remove floating point operations.

- Inline subroutine calls.
- Don't allocate new memory blocks unnecessarily.
- Fold constants together.
- Move I/O into a buffer.
- Try not to divide.
- Try not to do expensive casts.
- Move a pointer rather than recomputing indices.

2.7 How to Deal with I/O Expense

For a lot of problems, processors are fast compared to the cost of communicating with a hardware device. This cost is usually abbreviated I/O, and it can include network cost, disk I/O, database queries, file I/O, and other things that tend to make some hardware not very close to the processor do something. Therefore building a fast system is often more a question of improving I/O than improving the code in some tight loop, or even improving an algorithm.

There are two very fundamental techniques to improving I/O: caching and representation. Caching is avoiding I/O (generally avoiding the reading of some abstract value) by storing a copy of that value locally so no I/O is performed to get the value. It brings with it cache coherency problems, but can be very effective and a lot has been written about it. Representation is the approach of making I/O cheaper by representing data more efficiently. This is often in tension with other demands, like human readability and portability.

Representations can often be improved by a factor of two or three from their first implementation. Techniques for doing this include using a binary representation instead of a human readable one, transmitting a dictionary of symbols along with the data so that long symbols don't have to be encoded, and, at the extreme, things like Huffman encoding. I think this often represents a form of low-hanging fruit.

A third technique that is sometimes possible is to improve the locality of reference by pushing the computation closer to the data. For instance, if you are reading some data from a database and computing something simple from it, such as a summation, try to get the database server to do it for you. This is highly dependent on the kind of system you're working with, but in general building a server to produce computation results close to the data so that only small amounts of data have to be transferred should be explored more often.

2.8 How to Manage Memory

Memory is a precious resource that you can't afford to run out of. You can often ignore it for a while but eventually you will have to decide how to manage memory.

Space that needs to persists beyond the scope of a single subroutine is often called *heap allocated*. Depending on the system you use, you may have to explicitly deallocate such space when it is about to become garbage or you may rely on garbage collector. A chunk of memory is garbage when nothing refers to it. A garbage collector notices garbage and frees its space without any action required by the programmer. Garbage collection is wonderful. It lessens errors and increases code brevity and concision cheaply. Use it when you can. But even with garbage collection, you can fill up all memory with garbage. A classic mistake is to use a hash table as a cache and forget to remove the references in the hash table. Since the reference remains, the referent is uncollectable but useless. This is called a memory leak. You should look for and fix memory leaks early. If you have long running systems memory may never be exhausted in testing but will be exhausted by the user.

The creation of new objects is moderately expensive on any system. Memory allocated directly in the local variables of a subroutine is not because the policy for freeing it can be very simple. You should avoid unnecessary object creation and unnecessary allocation of memory outside of local subroutine parameters.

An important case occurs when you can define an upper bound on the number of objects you will need at one time. If these objects all take up the same amount of memory, you may be able to allocate a single block of memory, or a buffer, to hold them all. The objects you need can be allocated and released inside this buffer in a set rotation pattern, so it is sometimes called a ring buffer. This is usually faster than heap allocation.

Sometimes you have to explicitly free allocated space so it can be reallocated rather than rely on garbage collection. Then you must apply careful intelligence to each chunk of allocated memory and design a way for it to be deallocated at the appropriate time. The method may differ for each kind of object you create. You must make sure that every execution of a memory allocating operation is matched by a memory deallocating operation eventually. This is so difficult that people often simply implement a rudimentary form or garbage collection, such as reference counting, to do this for them.

2.9 How to Deal with Intermittent Bugs

The intermittent bug is a cousin of the 50-foot-invisible-scorpion-from-outer-space kind of bug. This nightmare occurs so rarely that it is hard to observe, yet often enough that it can't be ignored. You can't debug because you can't find it.

The intermittent bug has to obey the same laws of logic everything else does. What makes it hard is that it occurs only under unknown conditions. Try to record the circumstances under which it does occur, so that you can guess at what the variability really is. The condition may be related to data values, such as "This only happens when we enter *Wyoming* as a value." If that is not the source of variability, the next suspect should be improperly synchronized concurrency.

Try, try, try to reproduce it. If you can't reproduce it, set a trap for it by building a logging system, a special one if you have to, that can log what you guess is what you need when it really occurs. If the bug only occurs in production, this is a long process. Each hint that you get from the log about what the problem may be may not provide the solution but suggest the need for more logging. The new logging may take a long time to be put into production. Then you have to wait for the bug to occur to get more information. This cycle can go on for some time.

2.10 How to Learn Design Skills

To learn how to design software, study the action of a mentor by being physically present when they are designing. Then study well-written pieces of software. After that, you can read some books on the latest design techniques.

Then you must do it yourself. Start with a small project. When you are finally done, consider how the design failed or succeeded and how you diverged from your original conception. The move on to larger projects, hopefully in conjunction with other people. Design is a matter of judgment that takes years to acquire. A smart programmer can learn the basics adequately in two months and can improve from there.

It is natural and helpful to develop your own style, but remember that design is an art, not a science. People who write books on the subject have a vested interest in making it seem scientific. Don't become dogmatic about particular design styles.

3 Team Skills

3.1 Why estimation is important

To get a working software system in active use as quickly as possible requires planning the development, but also planning the documentation, deployment, marketing, sales, and finance. Without predictability of the development time, it is impossible to plan these effectively.

Good estimation provides predictability. Managers love it, as well they should. The fact that it is impossible, both theoretically and practically, to predict accurately how long it will take to develop software if often lost on managers. We are asked to do this impossible thing all the time, and we must face up to it honestly. However, it would be dishonest not to admit this impossibility, and when necessary, explain it. There is a lot of room for miscommunication about estimates, as people have a startling tendency to think wishfully that the sentence:

> I estimate it might be possible if I really understand that problem that it is about 50% likely to be completed in 5 weeks if no one bothers us in that time.

really means:

> I promise to have it all done 5 weeks from now.

Therefore explicitly discuss what the estimate means with the person you give it to, as if they were a simpleton. Restate your assumptions, no matter how obvious they seem to you.

3.2 How to Estimate Programming Time

Estimation takes practice. It also takes labor. It takes so much labor it may be a good idea to estimate the time it will take to make the estimate, especially if you are asked to estimate something you consider stupid.

When asked to provide an estimate of something big, the most honest thing to do is to stall. Most engineers are enthusiastic and eager to please, and stalling certainly will displease the stalled. But an on-the-spot estimate probably can't be accurate and honest.

While stalling, it may be possible to consider doing or prototyping the task. If political pressure permits, this is the most accurate way of producing the estimate, and makes real progress.

When not possible, you should first establish the meaning of the estimate very clearly. Restate that meaning as the first and last part of your written estimate. Prepare a written estimate by decomposing the task into progressively smaller subtasks until each small task is no more than a day, and ideally half a day at most in length. The most important thing is not to leave anything out. For instance, documentation, testing, time for planning, time for communicating with other groups, and vacation time are all very important. If you spend part of each day dealing with knuckleheads, put a line item for that in the estimate. This gives your boss visibility into what is using up your time at a minimum, and might get you more time.

I know good engineers who pad estimates implicitly, but I recommend that you do not. One of the results of this is to decrease trust somewhat. For instance, an engineer might estimate 3 days for a task that he or she truly thinks will take one day. The engineer may plan to spend 2 days documenting it, or two days working on some other useful project. But it will be detectable that the task was done in only one day (if it turns out that way), and the appearance of slacking or overestimating will be created. It's far better to give proper visibility into what one is actually doing. If documentation takes twice as long as coding and the estimate says so, tremendous advantage is gained by making this visible to the manager.

Pad explicitly instead. If a task will probably take one day but might take ten days if your approach doesn't work, note this somehow in the estimate if you can; if not, at least do an average weighted by your estimates of the probabilities. Any risk factor that you can identify and assign an estimate to should go into the schedule. One person is unlikely to be sick in any given week. But a large project with many engineers will have some sick time; likewise vacation time. And what is the probability of a mandatory company-wide training seminar? If it can be estimated, stick it in. There are of course, unknown unknowns, or unk-unks. Unk-unks by definition cannot be estimated individually. You can try to create a global line item for all unk-unks, or handle them in some other way that you communicate to your boss. You cannot, however, let your boss forget that they exist, and it is deucedly easy for an estimate to become a schedule without the unk-unks considered.

In a team environment, you should try to have the people who will do the work do the estimate, and you should try to have team-wide consensus on estimates. People vary widely in skill, experience, preparedness, and confidence. Calamity strikes if a strong programmer estimates for herself and then weak programmers are held to this estimate. The act of having the whole team agree on a line-by-line basis to the estimate clarifies the team understanding, as well as allowing the opportunity for tactical reassignment of resources (for instance, shifting burden away from weaker team members to stronger.)

If there are big risks that cannot be evaluated, it is your duty to state so forcefully enough that your manager does not commit to them and then become embarassed when the risk occurs. Hopefully in such a case whatever is needed will be done to decrease the risk.

If you can convince your company to use Extreme Programming, you will only have to estimate relatively small things, and this is both more fun and more productive.

3.3 How to Find Information

The nature of the what you need to know determines how you should find it.

If you need information about concrete things that is objective and easy to verify, for example the latest patch level of a software product, ask a large number of people politely by searching the internet for it or by posting on a discussion group. Anything that smacks of either opinion or subjective interpretation should not be searched for on the internet, as the ratio to drivel of truth is too low.

If you need general knowledge about something subjective, the history of what people have thought about it, go to the library (the physical building in which books are stored.) For example, to learn about math or mushrooms or mysticism, go to the library.

If you need to know how to do something that is not trivial, get two or three books on the subject and read them. You might learn how to do something trivial, like install a software package, from the Internet. You can even learn important things like good programming technique, but you can easily spend more time searching and sorting the results and attempting to divine the authority of the results than it would take to read the pertinent part of a solid book.

If you need information that no one else could be expected to know, for example, does this software that is brand new work on gigantic data sets, you must still search the internet and the library. After those options are completed exhausted, you may design an experiment to ascertain it, such as trying it.

If you want an opinion or a value judgment that takes into account some unique circumstance, talk to an expert. For instance, if you want to know if it is a good idea to build a modern database management system in LISP, you should talk to a LISP expert and a database expert.

If you want to know how likely it is that a faster algorithm for a particular application exists that has not yet been published, talk to someone working in that field.

If you want to make a personal decision that only you can make, such as whether or not you should start a business, then consider the Franklin method or divination. Suppose you have studied the idea from all angles, have done all you homework, and worked out all the consequences and pros and cons in your mind, and yet still remain indecisive. The multitude of available divination techniques are very useful for determining your own semi-concious desires, as the each present a complete ambiguous and random pattern that your own subconcious will assign meaning to.

3.4 How to Utilize People as Information Sources

Respect every person's time, and balance it against your own. Asking someone a question accomplishes far more than just receiving the answer. The person learns about you, both by enjoying your presence and hearing the particular questions. You learn about the person in the same way, and you may learn the answer you seek. This is usually far more important than your question.

However, the value of this diminishes the more you do it. You are, after all, using the most precious commodity a person has, their time. The benefits of communication must be weighed against the costs. Furthermore, the particular costs and benefits derived differ from person to person. I strongly believe that an executive of one hundred people should spend five minutes a month talking to each person in her organization, which would be about five per cent of her time. But ten minutes might be too much, and five minutes is too much if they have one thousand employees. The amount of time you spend talking to each person in your organization depends on their role (more than their position). You should talk to your boss more than your boss's boss, but you should talk to your boss's boss a little. It may be uncomfortable, but I believe you have a duty to talk a little bit to all your superiors each month no matter what.

The basic rule is that everyone benefits from talking to you a little bit, and the more they talk to you the less benefit they derive. It is your job to provide them this benefit, and to get the benefit of communicating with them, keeping the benefit in balance with the time spent.

It is important to respect your own time. If talking to someone, even if it will cost them time, will save you a great deal of time, then you should do it unless you think their time is more valuable than yours to the tribe by that factor.

A strange example of this is the summer intern. A summer intern in a highly technical position can't be expected to accomplish too much; they can be expected to pester the hell out of everybody there. So why is this tolerated?

Because the pestered are receiving something important from the intern. They get a chance to showoff a little. They get a chance to hear some new ideas, maybe; they get a chance to see things from a different perspective. They may also be trying to recruit the intern, but even if that is not to happen they gain a lot of benefit.

You should ask people for a little bit of their wisdom and judgment whenever you honestly believe they have something to say. This flatters them and you will learn something and teach them something. A good programmer does not often need the advice of a Vice President of Sales, but if she ever does, she should be sure to ask for it.

3.5 How to Document Wisely

Life is too short to write crap nobody will read. If you write crap, nobody will read it. Therefore a little good documentation is best. Bad documentation is very bad. Managers often don't understand this, because even bad documentation gives them a false sense of security that they are not dependent on their programmers. If someone absolutely insists that you write truly useless documentation, say yes and quietly begin looking for a better job.

There's nothing quite as effective as putting an accurate estimate of the amount of time it will take to produce good documentation into an estimate to slacken the demand for documentation. The truth is cold and hard: documentation, like testing, can take many times longer than developing code.

Writing good documentation is, first of all, good writing. I suggest you find books on writing, study them, and practice. But even if you are a lousy writer or have poor command of the language in which you must document, the Golden Rule is all you really need: "Do unto others as you would have them do unto you." Take time to really think about who will be reading your documentation, what they need to get out of it, and how you can teach that to them. If you do that, you will be an above average documentation writer, and a good programmer.

When it comes to actually documenting code itself, as opposed to producing documents that can actually be read by non-programmers, the best programmers I've ever known hold a universal sentiment: write self-explanatory code and don't document code except in the places that you cannot make it clear. There are two good reasons for this. First, anyone who needs to see code-level documentation will in most cases be able to and prefer to read the code anyway. Admittedly, this seems easier to the experienced programmer than to the beginner. More importantly however, is that the code and the documentation cannot get our of sync if there is no documentation. The source code can at worst be wrong and confusing. The documentation, if not written perfectly, can lie, and that is a thousand times worse.

This does not make it easier on the responsible programmer. How does one write self-explanatory code? What does that even mean? It means:

- writing code knowing that someone will have to read it;

- applying the golden rule;
- using whatever rules of good writing you may have learned;
- choosing a solution that is straightforward, even if you could get by with another solution faster;
- sacrificing small optimizations that obfusticate the code;
- thinking about the reader and spending some of your precious time to make it easier on her.

3.6 How to Work with Poor Code

It is very common to have to work with poor quality code that someone else wrote. Don't think too poorly of them, however, until you have walked in their shoes. They may have been asked very consciously to get something done quickly to meet schedule pressure. But in order to work with unclear code you must understand it. To understand it takes learning time, and that time will have to come out of some schedule, somewhere, and you must insist on it. To understand it, you will have to read the source code. You will probably have to experiment with it.

This is a good time to document, even if it is only for yourself, because the act of trying to document the code will force you to consider angles you might not have considered, and the resulting document may be useful. While you're doing this, consider what it would take to rewrite some or all of the code. Would it actually save time to rewrite some of it? Could you trust it better if you rewrote it? Be careful of arrogance here. If you rewrite it, it will be easier for you to deal with, but will it really be easier for the next person who has to read it? If you rewrite it, what will the test burden be? Will the need to re-test it outweigh any benefits that might be gained?

In any estimate that you make for work against code you didn't write, the quality of that code should affect your perception of the risk of problems and unk-unks.

It is important to remember that abstraction and encapsulation, two of the programmers best tools, are particularly applicable to lousy code. You may not be able to redesign a large block of code, but if you can add a certain amount of abstraction to it you can obtain some of the benefits of a good design without reworking the whole mess. In particular, you can try to wall off the parts that are particularly bad so that they may be redesigned independently.

3.7 How to Use Source Code Control

Source code control systems let you manage projects effectively. They're very useful for one person, and essential for a group. The track all changes in different versions so that no code is ever lost and meaning can be assigned to changes. I was late to appreciate the benefits of them but now I wouldn't live without one even on a one-person project.

A good technique for using a source code control system is to stay within a few days of being up to date at all time. Code that can't be finished in a few days is checked in, but in such a way that is inactive and will not be called, and hence not create any problem for anybody else, until it has been unit tested.

3.8 How to Unit Test

Unit testing, the testing of an individual piece of coded functionality by the team that wrote it, is a part of coding, not something different from it. Part of designing the code is designing how it will be tested. You should write down a test plan, even if it is only one sentence. Sometimes the test will be simple: "Does the button look good?" Sometimes it will be complex: "Did this matching algorithm return precisely the correct matches?"

Use assertion checking and test drivers whenever possible. This not only catches bugs early, but is very useful later on and lets you eliminate mysteries that you would otherwise have to worry about.

The Extreme Programming developers are writing extensively on unit testing effectively; I can do no better than to recommend their writings.

3.9 How to Stress Test

Unlike unit testing, stress testing is fun. At first it appears that the purpose of stress testing is to find out if the system works under a load. In reality, it is common that the system does work under a load but fails to work in some way when the load is heavy enough. I call this hitting the wall or *bonking*[1]. There may be some exceptions, but there is almost always a wall. The purpose of stress testing is to figure out where the wall is, and then figure out how to move the wall further out.

A plan for stress testing should be developed early in the project, because it often helps to clarify exactly what is expected. Is two seconds for a web page request a miserable failure or a smashing success? Is five hundred concurrent users enough? That of course depends, but one must know the answer when designing the system that answers the request. The stress test needs to model reality well enough to be useful. It isn't really possible to simulate five hundred human using a system concurrently very easily, but one can at least create five hundred simulations and try to model some part of what they might do.

In stress testing, start out with a light load and load the system along some dimension such as input rate or input size until you bonk. If the wall is too close to satisfy your needs, figure out which resource is the bottleneck (there is usually a dominant one.) Is it memory, processor, I/O, network bandwidth, or data contention? Then figure out how you can move the wall. Note that moving the wall, that is, increase the maximum load the system can handle, might not help or might actually hurt the performance of a lightly loaded system. Usually,

[1]This term has several meanings, derived from "to hit", but is in particular used by athletes to describe running out of blood sugar or some other basic resource that manifests as a sudden rather than gradual degradation of performance or spirt.

performance under heavy load is more important than performance under a light load.

You may have to get visibility into several different dimensions to build up a mental model of it; no single technique is sufficient. For instance, logging often gives a good idea of the wall clock time between two events in the system, but unless carefully constructed, doesn't give visibility into memory utilization or even data structure size. Similarly, in a modern system a number of computers and many software systems may be cooperating. Particularly when you are bonking (that is, the performance is non-linear in the size of the input) these other software systems may be a bottleneck. Visibility into these systems, even if only measuring the processor load on all participating machines can be very helpful.

Knowing where the wall is essential not only to moving the wall, but also to providing predictability so that the business can be managed effectively.

3.10 How To Recognize When To Break or Go Home

Computer programming is an activity, but it is also a culture. The unfortunate fact is that it is not a culture that values mental or physical health very much. For both cultural/historical reasons (the need to work at night on unloaded computers, for example) and because of overwhelming time-to-market pressure and the scarcity of programmers, computer programmers are overworked. I don't think you can trust all the stories you hear, but I think sixty hours a week is common, and fifty is pretty much a minimum. This means that often much more than that is required. This is serious problem for the good programmer, who is responsible not only for themselves and their teammates. You have to recognize when to go home, and sometimes when to suggest that other people go home. There can't be any fixed rules for solving this problem, anymore than there can be fixed rules for raising a child, for the same reason—every human being is different.

Beyond sixty hours a week is extraordinary effort for me, which I can apply for short periods of time (about one week), and that is sometimes expected of me. I don't know if it is fair to expect sixty hours of work from a person; I don't even know if forty is fair. I am sure however that it is stupid to work so much that you are getting little out of that extra hour you work. For me personally, that's any more than sixty hours a week. I personally think a programmer should exercise noblesse oblige and shoulder a heavy burden. However, it is not a programmer's duty to be patsy. The sad fact is programmers are often asked to be patsies in order to put on a show for somebody, for example a manager trying to impress an executive. Programmers often succumb to this because they are eager to please and not very good at saying no. There are four defenses against this:

- communicate as much as possible with everyone in the company so that no one can mislead the executives about what is going on,

- learn to estimate and schedule defensively and explicitly and give everyone visibility into what the schedule is and where it stands,
- learn to say no, and say no as a team when necessary, and
- quit if you have to.

Most programmers are good programmers, and good programmers want to get a lot done. To do that, they have to manage their time effectively. There is a certain amount of mental inertia associated with getting warmed up to a problem and deeply involved in it. Many programmers find they work best when they have long, uninterrupted blocks of time in which to get warmed up and concentrate. However, people must sleep and perform other duties. Each person needs to find a way to satisfy both their human rhythm and their work rhythm. Each programmer needs to do whatever it takes to procure efficient work periods, such as reserving certain days in which only most critical meetings will be attended.

Since I have children, I try to spend evenings with them sometimes. The rhythm that works best for me is to work a very long day, sleep in the office or near the office (I have a long commute from home to work) then go home early enough the next day to spend time with my children before they go to bed. I am not comfortable with this, but it is the best compromise I have been able to work out. Go home if you have a contagious disease. You should go home if you are thinking suicidal thoughts. You should take a break or go home if you think homicidal thoughts for more than a few seconds. You should send someone home if they show serious mental malfunctioning or signs of mental illness beyond mild depression. If you are being tempted to be dishonest or deceptive in a way that you normally are not due to fatigue, you should take a break. Don't use cocaine or amphetamines to combat fatigue. Don't abuse caffeine.

3.11 How to Deal with Difficult People

You will probably have to deal with difficult persons. You may even be one. If you are the kind of person who has a lot of conflicts with coworkers and authority figures, you should cherish the independence this implies, but work on your interpersonal skills without sacrificing your intelligence or principles.

This can be very disturbing to most programmers who have no experience in this sort of thing, and whose previous life experience has taught them patterns of behavior that are not useful in the workplace. Difficult persons are often inured to disagreement, and they are less affected by social pressure to compromise than others. The key is to respect them the right amount, which is more than you will want to but not as much as they might want.

Programmers have to work together as a team. When disagreement arise, it must be resolved somehow, it cannot be ducked for long. Difficult people often are extremely intelligent and have something very useful to say. It is critical that one listen and understand the difficult person without prejudice caused by the person. A failure to communicate is often the basis of disagreement and it can sometimes be removed with great patience. Try to keep this communication cool and cordial, and don't accept any baits for greater conflict that may be offered. After a reasonable period of trying to understand, make a decision.

Don't let a bully force you to do something you don't agree with. If you are the leader do what you think is best. Don't make a decision for any personal reasons, and be prepared to explain the reasons for your decision. If you are a teammate with a difficult person don't let the leaders decision have any personal impact. If it doesn't go your way, do it the other way whole-heartedly.

Difficult people do change and improve. I've seen it with my own eyes, but it is very rare. However, everyone has transitory ups and downs.

One of the challenges that every programmer but especially leaders face is keeping the difficult person fully engaged. They are more prone to duck work and resist passively than others.

Part II
Intermediate

4 Personal Skills

4.1 How to Stay Motivated

It is a wonderful and surprising fact that programmers are highly motivated by the desire to create artifacts that are beautiful, useful, or nifty. This desire is not unique to programmers nor universal but it is so strong and common among programmers that it separates the them from others in most companies.

This has practical and important consequences. If programmers are asked to do something that is not beautiful, useful, or nifty, they will have low morale. There's a lot of money to be made doing ugly, stupid, and boring stuff; but in the end fun will make the most money for the company.

4.2 How to Tradeoff Time versus Space

You can be a good programmer without going to college, but you can't be a good intermediate programmer without knowing basic computational complexity theory. You might be able to intuit how to tradeoff time against space without it, but you will not have a firm basis for communicating with your colleagues without it.

Time (processor cycles) and space (memory) can be traded off against each other. Engineering is about compromise, and this is a fine example. It is not always systematic. In general, however, one can save space by encoding things more tightly, at the expense of more computation time when you have to decode them. One can save time by caching, that is, spending space to store a local copy of something, at the expense of having to maintain the consistency of the cache. One can sometimes save time by maintaining more information in a data structure. This usually cost a small amount of space but may complicate the algorithm a lot.

Improving the space/time tradeoff can often change one or the other dramatically. However, before you work on this you should ask yourself if what you are improving is really the thing that most needs to be improved in your system. It's fun to work on an algorithm, but you can't let that blind you to the cold hard fact that improving something that is not a problem will not make any noticeable difference, and will create a test burden.

Memory on modern computers appears cheap, because unlike processor time you can't see it being used until you bonk. But when you bonk, you bonk hard. There are also other hidden costs to using memory, such as your effect on other programs that must be resident, and the time to allocate and deallocate it. Consider this carefully before you trade away space to gain speed.

4.3 How to Balance Brevity and Abstraction

Abstraction is key to programming. One should choose how abstract one needs to be carefully. Beginning programmers in their enthusiasm often create more abstraction than is really useful. One sign of this is if you create classes that don't really contain any code and don't really do anything except serve to abstract something. The attraction of this is understandable but the value of code brevity must be measured against the value of abstraction. Occasionally one sees a gentle mistake made by enthusiastic idealists: at the start of the project a lot classes are defined that are wonderfully abstract and should cover every eventuality. As the project progresses and fatigue sets in, the code itself becomes messy. Function bodies become longer than they should be. The empty classes are a burden to document that is ignored when under pressure. The final result would have been better if the energy spent on abstraction had been spent on keeping things short and simple. I strongly recommend the article "Succinctness is Power" by Paul Graham[6].

There is a certain dogma associated with useful techniques such as information hiding and object oriented programming that are sometimes taken too far. These techniques let one code abstractly and anticipate change. I personally think, however, that one should not produce much speculative code. For example, it is an accepted style to hide an integer variable on an object behind mutators and accessors, so that the variable itself is not exposed, only the little interface to it. This does allow the implementation of that variable to be changed without affecting the calling code, and is perhaps appropriate to a library writer who must publish a very stable API. But I don't think the benefit of this outweighs the cost of the wordiness of it when my team is owns the calling code and hence can recode the caller as easily as the called. Four or five extra lines of code is a heavy price to pay for this speculative benefit.

Portability poses a similar problem. Should code be portable to a different computer, compiler, software system or whatever, or simply easily ported? I think a non-portable, short-and-easily-ported piece of code is better than a long portable one. It is relatively easy and certainly a good idea to confine non-portable code to designated areas, such as a class that makes database queries that are specific to a given DBMS.

4.4 How to Learn New Skills

Learning new skills, especially non-technical ones, is the greatest fun of all. Most companies would have better morale if they understood how much this motivate programmers.

Humans learn by doing. Book-reading and class-taking are useful. But could you have any respect for a programmer who had never written a program? To learn any skill, you have to put yourself in a forgiving position where you can exercise that skill. When learning a new programming language, try to do a small project it in before you have to do a large project. When learning to manage a software project, try to manage a small one first.

A good mentor is no replacement for doing things yourself, but is a lot better than a book. What can you offer a potential mentor in exchange for their knowledge? At a minimum, you should offer to study hard so their time won't be wasted.

Try to get your boss to let you have formal training, but understand that it often no better than the same amount of time spent simply playing with the new skill you want to learn. It is however, easier to ask for training than playtime in our imperfect world, even though a lot of formal training is just sleeping through lectures waiting for the dinner party.

If you lead people, understand how they learn and help them learn by assigning them projects that are the right size and that exercise skills they want to learn. Don't forget that the most important skills for a programmer are not the technical ones. Give your people a chance to play and practice courage, honesty, and communication.

4.5 How to do Integration Testing

Integration testing is the testing of the integration of various components that have been unit tested. Integration is expensive and it comes out in the testing. You must include time for this in your estimates and your schedule.

Ideally you should organize a project so that there is not a phase at the end where things have to be integrated. It is far better to gradually integrate things as they are completed over the course of the project. If it is unavoidable estimate it carefully.

5 Team Skills

5.1 How to Manage Development Time

To manage development time, maintain a concise and up-to-date project plan. A project plan is an estimate, a schedule, a set of milestones for marking progress, an assignment of your team or your own time to each task on the estimate. It should also include other things you have to remember to do, such as meeting with the quality assurance people, preparing documentation, or ordering equipment. If you are on a team, the project plan should be a consensual agreement, both at the start and as you go.

The project plan exists to help make decisions, not to show how organized you are. If the project plan is either too long or not up-to-date it will be useless for making decisions. In reality, these decisions are about individual persons. The plan and your judgment let you decide if you should shift tasks from one person to another. The milestones mark your progress. If you use a fancy project planning tool, do not be seduced into creating a big design up front for the project, but use it maintain concision and up-to-dateness.

If you miss a milestone, you should take immediate action such as informing your boss that the scheduled completion of that project has slipped by that amount. The estimate and schedule could never have been perfect to begin with; this creates the illusion that you might be able to make up the days you missed in the latter part of the project. You might. But it is just as likely that you have underestimated that part as that you have overestimated it. Therefore the scheduled completion of the project has already slipped, whether you like it or not.

Make sure you plan includes time for:

- internal team meetings,
- demos,
- documentation,
- scheduled periodic activities,
- integration testing,
- dealing with outsiders,
- sickness,
- vacations,
- maintenance of existing products, and
- maintenance of the development environment.

The project plan can serve as a way to give outsiders or your boss a view into what you or your team are doing. For this reason it should be short and up-to-date.

5.2 How to Manage Third Party Software Risks

A project often depends on software produced by organizations that it does not control. There are great risks associated with third party software that must be recognized by everyone involved.

Never, ever, rest any hopes on *vapor*. *Vapor* is any alleged software that has been promised but is not yet available. This is the surest way to go out of business. It is unwise to be skeptical of a software company's promise to release a certain product with a certain feature at a certain date; it is far wiser to ignore it completely and forget you ever heard it. Never let it be written down in any documents used by your company.

If third party software is not vapor, it is still risky, but at least it is a risk that can be tackled. If you are considering using third party software you should devote energy early to evaluating it. People might not like to hear that it will take two weeks or two months to evaluate each of three products for suitability. But it has to be done as early as possible. The cost of integrating cannot be accurately estimated without a proper evaluation.

Understanding the suitability of existing third party software for a particular purpose is very tribal knowledge. It is very subjective and generally resides in experts. You can save a lot of time if you can find those experts. Often a project will depend on a third party software system so completely that if the integration fails the project will fail. Express risks like that clearly in writing in the schedule. Try to have a contingency plan, such as another system that can be used or the ability to write the functionality yourself if the risk can't be removed early. Never let a schedule depend on vapor.

5.3 How to Manage Consultants

Use consultants, but don't rely on them. They are wonderful people and deserve a great deal of respect. They often know more about specific technologies and even programming techniques than programmers that work as employees of a company. The best way to use them is as educators in-house that can teach by example.

However, they cannot usually become part of the team in the same sense that regular employees are, if only because you may not have enough time to learn their strengths and weaknesses. Their financial commitment is much lower. They can move more easily. They may have less to gain if the company does well. Some will be good, some will be average and some will be bad, but usually your selection of consultants will not be as careful as your selection of employees, so you will get more bad ones.

If they're going to write code you must review it carefully as you go along. You cannot get to the end of the a project with the risk of a large block of code that has not been reviewed. This is true of all team members, really, but you will usually have more knowledge of your closer team members.

5.4 How to Communicate the Right Amount

Consider carefully the cost of a meeting. It costs its duration multiplied by the number of participants. Meetings are sometimes necessary, but smaller is usually better. The quality of communication in small meetings is better, and less time overall is wasted. If you are having a meeting and anyone is bored, that should be a sign that perhaps you should have organized smaller meetings.

Everything possible should be done to encourage informal communication. More useful work done during lunches with colleagues than during any other time. It is a shame that more companies to not recognize this fact and support it.

5.5 How to Disagree Honestly and Get Away with It

Disagreement is a great opportunity to make a good decision, but it should be handled delicately. Hopefully you feel that you have expressed your thoughts adequately and been listened to before the decision is made. In that case there is nothing more to say, and you should decide whether you will stand behind the decision even thought you disagree with it. If you can support this decision even though you disagree, say so. This shows how valuable you are because you are independent and not a yes-man, but respectful of the decision and a team player.

Sometimes a decision will be made that you disagree with when the decision makers did not have the full benefit of you opinion. You should then evaluate whether you should raise the issue on the basis of the benefit to the company or tribe. If it is a small mistake in your opinion, it may not be worth reconsidering. If it is a large mistake in you opinion, then of course you must present an argument.

Usually, this is not a problem. In some stressful circumstances and with some personality types this can lead to things being taken personally. For instance, some very good programmers lack the confidence needed to challenge a decision even when they have good reason to believe it to be wrong. In the worst of circumstances the decision maker is insecure and takes it as a personal challenge to her authority. It is best to remember in such circumstances that people react with the reptilian part of their brains. You should present your argument in private, and try to show how new knowledge changes the basis on which the decision was made.

Whether the decision is reversed or not, you must remember that probably no one will ever have a right to say "I told you so!" since the other decision will not have been fully explored.

6 Judgment

6.1 How to Tradeoff Quality Against Development Time

Software development is always a compromise between what the project does and getting the project done. But you may be asked to make a tradeoff of quality to speed the deployment of a project that is offends your engineering sensibilities in the same way that making a toaster that is designed to wear out is offensive. For example, you may be asked to do something that is a poor software engineering practice and will lead to a lot of maintenance problems.

If this happens your first responsibility is to inform your team and to clearly explain the cost of the decrease in quality. After all, your understanding of it should be much better than your boss's understanding. Make it clear what is being lost and what is being gained, and how at what cost the lost ground will be regained in the next cycle. In this the visibility provided by a good project plan should be helpful. If the quality tradeoff affects the quality assurance effort point that out (both to your boss and quality assurance people.) If the quality tradeoff will lead to more bugs being reported after the quality assurance period, point that out.

If she still insists you should try to isolate the shoddiness into particular components that you can plan to rewrite or improve in the next cycle. Explain this to your team so that they can plan for it.

6.2 How to Manage Software System Dependence

Modern software systems tend to depend on a large number of components that are not directly under your control. This increases productivity through synergy and reuse. However, each component brings with it some problems:

- How will you fix bugs in the component?
- Does the component restrict you to particular hardware or software systems?
- What will you do if the component fails completely?

It is always best to encapsulate the component in some way so that it is isolated and so that it can be swapped out. If the component proves completely unworkable, you may be able to get a different one, but you may have to write your own. Encapsulation is not portability, but it makes porting easier and that is almost as good.

Having the source code for a component decreases the risk by a factor of four. With it, you can evaluate it easier, debug it easier, find workarounds easier, and make fixes easier. If you make fixes, you should give them to the owner of the component and try to get them to incorporate them into an official release; otherwise you will have to maintain an unofficial version uncomfortably.

6.3 How to Decide If Software Is Too Immature

The use of software other people wrote is one of the most effective ways to build quickly a solid system. It should not be discouraged, but the risks associated with it must be examined. One of the biggest risks is the period of bugginess and near inoperability that is often associated with software before it matures through usage into a usable product. When considering integrating with or becoming dependent in some way on a software system, whether created in house or by a third party, it is very important to consider if it is really mature enough to be used. Here are some questions you should ask yourself about it:

- Is it vapor? (Promises are very immature).
- Is there an accessible body of lore about the software?
- Are you the first user?
- Is there a strong incentive for continuation?
- Has it had a maintenance effort?
- Will it survive defection of the current maintainers?
- Is there a seasoned alternative at least half as good?
- Is it known to your tribe or company?
- Is it desirable to your tribe or company?
- Can you hire people to work on it even if it is bad?

6.4 How to Make a Buy vs. Build Decision

An entrepreneurial company that is trying to accomplish something with software has to make buy versus build decisions constantly. This requires a great combination of business, management, and engineering savvy. It should perhaps be called a buy and integrate vs. build and integrate decision because the cost of integration must be considered.

- How well do your needs match those for which it was designed?
- What portion of what you buy will you need?
- What is the cost of evaluating the integration?
- What is the cost of integration?
- Will buying increase or decrease long term maintenance costs?
- Will building put you in a business position you don't want to be in?

You should think twice before building something that is big enough to serve as the basis for an entire other business. Such ideas are often proposed by bright and optimistic persons that will have a lot to contribute to your team. If their idea is compelling, you may wish to change your business plan; but do not invest in a solution bigger than your own business without concious thought.

After considering these questions, you should perhaps prepare two draft project plans, one for building and one for buying. This will force you to consider the integration costs. You should also consider the long term maintenance costs of both solutions. To estimate the integration costs, you will have to be able to do a thorough evaluation of the software before you buy it. If you can't evaluate it you will assume an unreasonable risk in buying it and you should decide against buying that particular product. If there are several buy decisions under consideration, some energy will have to be spent evaluating each.

6.5 How to Grow Professionally

Assume responsibility in excess of your authority. Play the role that you desire. Express appreciation for people's contribution to the success of the larger organization, as well as things as that help you personally.

If you want to become a team leader, instigate the formation of consensus. If you want to become a manager, take responsibility for the schedule. You can usually do this comfortably while working with a leader or a manager, since this frees her up to take greater responsibility. If that is too much to try, do it a little at a time.

Evaluate yourself. If you want to become a better programmer, ask someone you admire how you can become like her. You can also ask your boss, who will know less but have a greater impact on your career.

Plan ways to learn new skills, but the trivial technical kind like learning a new software system and the hard social kind like writing well, by integrating them into your work.

6.6 How to Evaluate Interviewees

Evaluating potential employees is not given the energy it deserves. A bad hire, like a bad marriage, is terrible. A significant portion of everyone's energy should be devoted to recruitment, but this is rarely done.

There are different interviewing styles. Some are torturous, designed to put the candidate under a great deal of stress. This serves a very valuable purpose of possibly revealing character flaws and weaknesses under stress. Candidates are no more honest with interviewees than they are with themselves, and the human capacity for self-deception is astonishing.

You should at a minimum give the candidate the equivalent of an oral examination on the technical skills for two hours. With practice, you will be able to quickly cover what they know and quickly retract from what they don't know to mark out the boundary. Interviewees will respect this. I have several times heard with my own ears them say that the was one of their motivations for choosing a company. Good ones want to be hired for their skills, not where they worked last or what school they went to or some other inessential characteristic.

In doing this, you should also evaluate their ability to learn, which is far more important than what they know. You should also watch for the whiff of brimstone that is given off by difficult people. You can often recognize it later, but in the heat of the interview it is hard to recognize. How well people communicate and work with people is more important than being up on the latest programming language.

Finally, interviewing is also a process of selling. You should be trying to sell your company to the candidate. However, you are talking to a programmer, so normal salesmanship does not work. Don't try to color the truth. Start off with the bad stuff, then finish strong with the good stuff.

6.7 How to Know When to Apply Fancy Computer Science

There is a body of knowledge about algorithms, data structures, mathematics, and other gee-whiz stuff that most programmers know about but rarely use. In practice, this wonderful stuff is too complicated and generally unnecessary. There is no point in improving an algorithm when most of your time is spent making inefficient database calls, for instance. An unfortunate amount of programming consists of getting systems to talk to each other and using very simple data structures to build a nice user interface.

When is high technology the appropriate technology? When should one crack a book to get something other than a run-of-the-mill algorithm? It is sometimes useful to do this but it should be evaluated carefully.

The three most import considerations for the potential computer science technique are:

- Is it well encapsulated so that the risk to other systems is low and the overall increase in complexity and maintenance cost is low?
- Is the benefit startling (for example, a factor of two in a mature system or a factor of ten in a new system)?
- Will you be able to test and evaluate it effectively?

If a well-isolated algorithm that uses a slightly fancy algorithm can decrease hardware cost or increase performance by a factor of two across an entire system, then it would be criminal not to consider it. One of the keys to arguing for such an approach is to show that the risk is really quite low, since the proposed technology has probably been well studied, the only issue is the risk of integration. Here a programmer's experience and judgment can truly synergize with the fancy technology to make integration easy.

By carefully designing the encapsulating so that the risk is low, you should be able to produce an estimate with confidence that allows the costs and benefits of the proposal be properly judged.

6.8 How to Talk to Non-Engineers

Engineers and programmers in particular are recognized by popular culture as being different from other people. This implies that other people are different from us. This is worth bearing in mind when communicating with them. One should always understand the audience.

Non-engineers are smart, but not as grounded in creating technical things as we are. We make things. They sell things and handle things and count things and manage things, but they are not experts on making things.

They are not as good at working together on teams as engineers are (there are not doubt exceptions.) Their social skills are generally as good as or better than engineers in non-team environments, but their work does not always demand that they practice the kind of intimate, precise communication and careful subdivisions of tasks that we do. Their teams are more like groups.

Non-engineers may be too eager to please and they may be intimidated by you. Just like us, they may say yes without really meaning it to please you or because they are a little scared of you, and then not stand behind their words.

Non-programmers can understand technical things but they do not have technical judgment. They do understand how technology works, but they cannot understand why a certain approach would take three months and another one three days. This represents a great opportunity to synergize with them.

When talking to your team you will without thinking use a sort of shorthand, an abbreviated language that is effective because you will have much shared experience about technology in general and your product in particular. This does not work with outsiders, though they will get better with practice. You will have to take things more slowly with them.

With your team, the basic assumptions and goals need not be restated often, and most conversation focuses on the details. With outsiders, it must be the other way around. They may not understand things you take for granted. Since you take them for granted and don't repeat them, you can leave a conversation with an outsider thinking that you understand each other when really there is a large misunderstanding. You should assume that you will be misunderstood and watch carefully to find this misunderstanding in them. Try to get them to summarize or paraphrase what you are saying to make sure they understand.

I love working with non-engineers. It provides great opportunities to learn and to teach. You can often lead by example, in terms of the clarity of your communication. Engineers are trained to bring order out of chaos, to bring clarity out of confusion, and non-engineers like this about us. Because we have technical judgment and can usually understand business issues, we can often find a simple solution to a problem. Often non-engineers propose solutions that they think will make it easier on us out of kindness and a desire to do the right thing, when in fact a much better solution exists which they could not see because of their lack of technical judgment.

Part III
Advanced

7 Technological Judgment

7.1 How to Tell the Hard From the Impossible

It is our job to do the hard and discern the impossible. ¿From the point of view of most working programmers what is called research is impossible, in that it cannot be predicted and estimated and scheduled. A large volume of *mere work* is hard, but not necessarily impossible.

The distinction is not facetious because you may very well be asked to do what is practically impossible, either from a scientific point of view or a software engineering point of view. It then becomes you job to help the entrepreneur find a reasonable solution which is *merely hard* and gets most of what she wanted. A solution is merely hard when it can be confidently scheduled and the risks are understood.

It is impossible to satisfy a vague requirement, such as “Build a system that will compute the most attractive hair style and color for any person.” If the requirement can be made more crisp it will often become merely hard, such as “Build a system to compute an attractive hair style and color for a person, allow them to preview it and make changes, and have the customer satisfaction based on the original styling be so great that we make a lot of money.” If there is not crisp definition of success, you will not succeed.

7.2 How to Utilize Embedded Languages

Embedding a programming language into a system has an erotic fascination to a programmer. It makes the system tremendously powerful. It allows her to exercise her most creative and promethean skills. It makes the system into a friend for her.

I and many other programmers have fallen into the trap or creating special purpose embedded languages. I fell into it twice. The problem is that a programmable system is wonderful only if you are a programmer.

It does unquestionably offer tremendous power. The best text editors in the world all have embedded languages. This can be used to the extent that the intended audience can master the language. Of course, use of the language can be made optional, as it is in text editors, so that initiates can use it and no one else has to.

The real question to ask oneself before embedding a language is: Does this work with or against the culture of my audience? If you intended audience is exclusively non-programmers, how will it be helping? If you intended audience is exclusively programmers, might they not prefer an applications programmers interface? And what language will it be? Programmers don’t want to learn a new language that is outside their culture just for fun; but if it meshes with their culture they will not have to spend much time learning it. It is a joy to create a new language. But we should not let that blind us to the needs of the user. Unless you have some truly original needs and ideas, why not use some existing language so that you can leverage the familiarity users already have with that language?

7.2.1 Choosing Languages

The solitary program that loves his work (a hacker) can choose the best language for the task. Mosting working programmers have very little control of the language they will use. Generally, this issue is dictated by pointy-haird bosses who are making a politcal decision, rather than a technological decision, and lack the courage to say promote an unconventional tool even when they know, often with firsthand knowledge, that the less accepted tool is best. In other cased the very real benefit of unity among the team, and to some extent with a larger community, precludes choice on the part of the individual.

8 Compromising Wisely

8.1 How to Fight Schedule Pressure

Time-to-market pressure is the pressure to deliver a good product quickly. It is good because it reflects a financial reality, and is healthy up to a point. Schedule pressure is the pressure to deliver something faster than it can be delivered and it is wasteful, unhealthy and all too common.

Schedule pressure exists for several reasons. The people who tasks programmers do not fully appreciate what a strong work ethic we have and how much fun it is to be a programmer. Perhaps because they project their own behavior onto us, they believe that asking for it sooner will make us work harder to get it there sooner. This is probably actually true but the effect is very small, and the damage is very great. Additionally, they have no visibility into what it really takes to produce software. Not being able to see it, and not be able to create it themselves, the only thing they can do is see time-to-market pressure and fuss at programmers about it.

The key to fighting schedule pressure is simple to turn it into to time-to-market pressure. The way to do this to give visibility into the relationship between the available labor and the product. Producing an honest, detailed, and most of all understandable estimate of all the labor involved is the best way to do this. It has the added advantage of allowing good management decisions to be made about possible functionality tradeoffs.

The key insight that the estimate must make plain is that labor is an almost incompressible fluid. You can't pack more into a span of time anymore than you can pack more water into a container than that container's volume. In a sense, it is not a programmer's job to say no, but rather to say "What will you give up to get that thing you want?" The effect of producing such estimates will be to increase the respect for programmers. This is how other professionals behave. Programmers's hard work will be visible. Setting an unrealistic schedule will also be painfully obvious to everyone. Programmers cannot be hoodwinked. It is disrespectful and demoralizing to ask them to do something unrealistic. Extreme Programming aplifies this and builds a process around it; I hope that every reader will be lucky enough to you it.

8.2 How to Understand the User

It is your duty to understand the user, and to help your boss understand the user. Because the user is not as intimately involved in the creation of your product as you are, she behaves a little differently:

- The user generally makes short pronouncements.
- The user has her own job; she will mainly think of small improvements in your product, not big improvements.
- The user can't have a vision across all the users of your product.

It is your duty to give them what they really want, not what they say the want. It is however, better to propose it to them and get them to agree that your proposal is what they really want before you begin, but they may not have the vision to do this. Your confidence in your own ideas about this should vary. You must guard against both arrogance and false modesty in terms of knowing what the customer really wants. Programmers are trained to design and create. Market researchers are trained to figure out what people want. These two kinds of persons, or two modes of thought in the same person, working harmoniously together give the best chance of formulating the correct vision.

The more time you spend with users the better you will be able to understand or guess what will really be successful. You should try to test your ideas against them as much as you can. You should eat and drink with them if you can.

8.3 How to Get a Promotion

To be promoted to a role, act out that role first.

To get promoted to a title find out what is expected of that title and do that.

To get a pay raise, negotiate armed with information.

If you feel like you are past due for a promotion, talk to your boss about it. Ask her explicitly what you need to do to get promoted, and try to do it. This sounds trite, but often your perception of what you need to do and your boss's will differ considerably. Also this will pin your boss down in some ways.

9 Serving Your Team

9.1 How to Develop Talent

Nietschze exaggerated when he said:

> What does not destroy me, makes me stronger.

Your greatest responsibility is to your team. You should know each of them well. You should stretch your team, but not overburden them. You should usually talk to them about the way in which they are being stretched. If they buy in to it, they will be well motivated. On each project or every other project try to stretch them in both a way that they suggest and a way that you think will be good for them. Stretch them not by giving them more work, but by giving them a new skill or better yet a new role to play on the team.

You should allow people to fail occasionally and plan for some failure in your schedule. If there is never any failure, there can be no sense of adventure. If there are not occasional failures, you are not trying hard enough. When someone fails, you should be as gentle as you can with her but don't treat it as if she had succeeded.

Try to get each team member to buy in and be well motivated. Ask each of them explicitly what they need to be well-motivated if they are not. You may have to leave them dissatisfied, but you should know what everybody desires.

You can't give up on someone who is intentionally not carrying their share of the load because of low morale or dissatisfaction and just let them be slack. You must try to get them well-motivated and productive. As long as you have the patience, keep this up. When your patience is exhausted, fire them. You cannot allow someone who is intentionally working below their level to remain on the team, since it is not fair to the team.

Make it clear to the strong members of your team that you think they are strong by saying so in public. Praise should be public and criticism private.

The strong members of the team will naturally have more difficult tasks than the weak members of the team. This is perfectly natural and nobody will be bothered by it so long as everyone works hard.

It is an odd fact that is not reflected in salaries that a good programmer is more productive than ten bad programmers. This creates a strange situation. It will often be true that you could move faster if your weak programmers would just get out of the way. If you did this you would in fact make more progress in the short term. However, your tribe would lose some important benefits, name the training of the weaker members, the spreading of tribal knowledge, and the ability to recover from the loss of the strong members. The strong must be gentle in this regard and consider the issue from all angles.

You can often give the stronger team members challenging, but carefully delineated, tasks.

9.2 How to Choose What to Work On

You balance your personal need against the needs of the team in choosing what aspect of a project to work on. You should do what you are best at, but try to find a way to stretch yourself not by taking on more work but by exercising a new skill. Leadership and communication skills are more important than technical skills. If you are very strong, take on the hardest or riskiest task, and do it as early as possible in the project to decrease risk as much as possible.

9.3 How to Get the Most From Your Teammates

To get the most from your teammates, develop a good team spirit and try to keep every individual both personally challenged and personally engaged.

To develop team spirit, corny stuff like logoized clothing and parties are good but not as good as personal respect. If everyone respects everyone else, nobody will want to let anybody down. Team spirit will be created when people make sacrifices for the team and they think in terms of the good of the team rather than their own personal good. As a leader, you can't ask for more than you give yourself.

One of the keys to team leadership is to facilitate consensus so that everyone has buy in. This occasionally means allowing your teammates to be wrong. That is, if it does not harm the project too much, you must let some of your team do things a certain way if they have consensus on it even if you know with great confidence it is the wrong thing to do. You don't always have to do this, but you should do it some. When this happens, don't agree, simply disagree openly and accept the consensus. Don't sound hurt, or like you're being forced into it, simply state that you disagree but think the consensus of the team is more important. This will often cause them to backtrack. Don't insist that they go through with their initial plan if they do backtrack.

If there is an individual who cannot be brought into consensus after you have discussed the issues from all appropriate sides, simply assert that you have to make a decision and that is what your decision is. If there is a way to judge if your decision will be wrong or if it is later shown to be wrong, switch as quickly as you can and recognize the persons who were right.

Ask your team, both as a group and individually, what they think would create team spirit and make for an effective team.

Praise often rather than lavishly. Especially praise those who disagree with you when they are praiseworthy. Praise in public and criticize in private; with one exception: sometimes growth or the correction of a fault can't be praised without drawing embarrassing attention to the original fault, so that growth should be praised in private.

9.4 How to Divide Problems Up

Its fun to take a software project and divide it up into tasks that will be performed by individuals. This should be done early. Sometimes managers would like to think that an estimate can be made without consideration of the individuals that will perform the work. But how can this be when we know individual productivity differs by an order of magnitude and productivity of different tasks by the same person differs by an order of magnitude?

Just as a composer usually considers the timbre of the instrument that will play a part, the experienced team leader will usually not be able to separate the division of the project into tasks from the team members to which they will be assigned. This is part of the reason that a high performing team should not be broken up.

There is a certain danger in this in that people will become bored as they build upon the strengths and never improve their weaknesses or learn new skills. However, specialization is a very useful productivity tool if not overused.

9.5 How to Gather Support for a Project

To gather support for a project, create and communicate a vision that demonstrates real value to the organization as a whole. Attempt to let others share in your vision creation. This gives them both a reason to support you and getting the benefit of their ideas. Individually recruit key supporters for your project. Wherever possible, show, don't tell. If possible construct a prototype or a mockup to demonstrate your ideas. A prototype is always powerful but in software it is far superior to any written description.

9.6 How to Grow a System

The seed of a tree contains the idea of the adult but does not fully realize the form and potency of the adult. The embryo grows. It becomes larger. It looks more like the adult, and has more of the uses. Eventually it bears fruit. Later on it dies and it bodies feeds other organisms.

Software is like that; we have the luxury of treating it like that. A bridge is not like that; there is never a baby bridge but merely an unfinished bridge.

It is good to think of software as growing, because it allows us to make useful progress before we have a perfect mental image. We can get feedback from users and use that to correct the growth. Pruning off weak limbs is healthful.

The programmer must design a finished system that can be delivered and used. But the advanced programmer must do more. She must design a growth path that ends in the finished system. It is her job to take a germ of an idea and build a path that takes that little idea as smoothly as possible into a useful artifact.

To do this, she must visualize the end result and communicate it in a way that the engineering team can get excited about. But she must also communicate to them a path that goes from wherever they are now to where the want to be with no large leaps. The tree must stay alive the whole time; it cannot be dead at one point and resurrected later.

This approach is captured in spiral development. Milestones that are never too far apart are used to mark progress along the path. In the ultra-competitive environment of business, it is best if the milestones can be released and make money as early as possible, even if they are far away from a well-designed endpoint. One of the programmer's jobs is to the balance immediate payoff against future payoff by wisely choosing a growth path expressed in milestones.

The advanced programmer has the triple responsibility of growing software, teams, and persons.

9.7 How to Communicate Well

To communicate well, you have to recognize how hard it is. It is a skill unto itself. It is made harder by the fact that the persons with whom you have to communicate are flawed. They do not work hard at understanding you. They speak poorly and write poorly. They are often overworked or bored, and at a minimum somewhat focused on their own work rather than the larger issues you may be addressing. One of the advantages of taking classes and practicing writing, public speaking, and listening is that if you become good at it you can more readily see where problems lie and how to correct them.

The programmer is a social animal whose survival depends on communication with her team. The advanced programmer is a social animal whose satisfaction depends on communication with people outside her team.

The programmer brings order out of chaos. One interesting way to do this is to initiate a proposal of some kind outside the team. This can be done in a *strawman* or em white-paper format or just verbally. This leadership has the tremendous advantage of setting the terms of the debate. It also exposes one to criticism, and worse, rejection and neglect. The advanced programmer must be prepared to accept this, because she has a unique power and therefore a unique responsibility. Entrepreneurs who are not programmers need programmers to provide leadership in some ways. Programmers are the part of the bridge between ideas and reality that rests on reality.

9.8 How to Tell People Things They Don't Want to Hear

You will often have to tell people things that will make them uncomfortable. Remember that you are doing this for a reason. Even if nothing can be done about the problem, you are telling them as early as possible so they will be well-informed.

The best way to tell someone about a problem is to offer a solution at the same time. The second best way is to appeal to them for help with the problem, which is usually flattering to them. If there is a danger that you won't be believed, you should gather some support for your assertion.

One of the most unpleasant and common things you will have to say is, "The schedule will have to slip." The conscientious programmer who bought into the schedule hates to say this, but must say it as early as possible. There is nothing worse than not taking action immediately when a milestone slips, even if the only action is to inform everyone. In doing this, it is better to do it as a team, at least in spirit if not physically. You will want your team's input on both where you stand and what can be done about it, and the team will have to face the consequences with you.

9.9 How to Deal with Managerial Myths

The term myth sometimes means fiction. But it has a deeper connotation. It also means a story of religious significance that explains the universe and mankind's relationship to it. Managers tend to forget what they learned as programmers and believe in certain myths. It would be as rude and unsuccessful to try to convince them these myths are false as to try to disillusion a devoutly religious person of their beliefs. For that reason, you should recognize these myths for what they are:

- More documentation is always better. (They want it, but they don't want you to spend any time on it.)
- Programmers can be equated. (Programmers vary by order of magnitude.)
- Resources can be added to a project to speed it. (The cost of communication with the new persons is more taxing than helpful.)
- It is possible to estimate software development reliably. (It is not even theoretically possible.)

Each of these myths reinforces the manager's idea that they have some actual control over what is going on. The truth is that managers facilitate if they are good, and impede if they are bad.

9.10 How to Deal with Organizational Chaos

There are often times of great organizational chaos. These are unsettling to everyone, but perhaps a little less unsettling to the programmer whose personal self-esteem is founded in her capacity rather than in her position. Organizational chaos is a great opportunity for programmers to exercise their magic power. I've saved this for last because it is a deep tribal secret. If you are not a programmer, please stop reading now.

> Engineers have the power to create and sustain.

Non-engineers can order people around but in a typical software company can create nothing on their own and only have the power that engineers grant them. They can create and sustain nothing without engineers. This power is proof against almost all the problems associated with organizational mayhem. When you have it you should ignore the chaos completely and carry on as if nothing is happening. You may of course get fired, but if that happens you can easily get a new job because of the magic power. More commonly, some stressed-out person who does not have the magic power will come into your cube and tell you to do something stupid. It is best to smile and nod until they go away and then carry on doing what you know is best for the company.

This course of action is the best for you personally, and the best for the company you work for. If you are a leader, tell your people to do the same thing and tell them to ignore what anybody other than yourself tells them, including your own superiors.

9.11 Request for Feedback

Please send me any comments you may have on this essay. It is a work-in-progress, and has received no editing, proofreading, or review of any kind.

Thank you.

Robert L. Read <read@hire.com>

A Glossary

- Unk-unk — (slang for unkown-unkown), the problems that cannot even be conceptualized at present that will steal time away from the project and wreck the schedule.
- Winner-take-allish — A competition is winner-take-allish if the reward is more based on the order of the relative success of the competitors rather than the actual success of the competitors. A race is winner-take-allish. Two homebuilders may compete with each other to build the best house, but if they loser is rewarded for the house she built anyway, their competition is not winner-take-allish.
- Printlining — the insertion of statements into a program on a strictly temporary basis that output information about the execution of the program for the purpose of debugging.
- Logging — the practice of writing a program so that it can produce a configurable output log describing its execution.
- Divide and Conquer — a technique of top-down design and more importantly of debugging that is the subdivision of a problem or a mystery into progressively smaller problems or mysteries.
- Vapor — illusionary and often deceptive promises of software that is not yet for sale and as often as not will never materialize into anything solid.
- Boss – the person who sets your tasks. In some cases, the user is the boss.
- Tribe — the people with whom you share loyalty to a common goal.
- Low-hanging fruit — big improvements that cost little.
- Garbage — objects that are no longer needed that hold memory.
- Entrepreneur — the initiator of projects.

References

[1] Kawasiki, Guy, Moreno, Michelle, and Kawasaki, Gary. 2000. Rules for Revolutionaries: The Capitalist Manifesto for Creating and Marketing New Products and Services. HarperBusiness.

[2] McConnell, Steve. 1996. Rapid Development: Taming Wild Software Schedules. Redmond, Wash.: Microsoft Press.

[3] McConnell, Steve. 1993. Code Complete. Redmond, Wash.: Microsoft Press.

[4] Beck, Kent. Extreme Programming Explained: Embrace Change.

[5] Beck, Kent and Fowler, Martin. Planning Extreme Programming.

[6] Graham, Paul. 2002. Articles on his website: http://www.paulgraham.com/articles.html. All of them, but especially Beating the Averages.

[7] Raymond, Eric Steven. 2002. How to Become a Hacker: http://www.tuxedo.org/ esr/faqs/hacker-howto.html

[8] Hunt, Andrew, Thomas, David, and Cunningham, Ward. The Pragmattic Programmer: From Journeyman to Master.