

Joel on Software

The Guerrilla Guide to Interviewing (version 3.0)

by Joel Spolsky

Wednesday, October 25, 2006

A motley gang of anarchists, free-love advocates, and banana-rights agitators have hijacked *The Love Boat* out of Puerto Vallarta and are threatening to sink it in 7 days with all 616 passengers and 327 crew members unless their demands are met. The demand? A million dollars in small unmarked bills, and a GPL implementation of WATFIV, that is, the esteemed Waterloo Fortran IV compiler. (It's surprising how few things the free-love people can find to agree on with the banana-rights people.)

As chief programmer of the Festival Cruise programming staff, you've got to decide if you can deliver a Fortran compiler from scratch in seven days. You've got a staff of two programmers to help you.

Can you do it?

"Well, I suppose, it depends," you say. One of the benefits of writing this article is that I get to put words into your mouth and you can't do a darn thing about it.

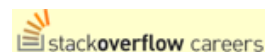
On what?

"Um, will my team be able to use UML-generating tools?"



[File a CV](#) and let the great jobs come to you!

Wanted: [Senior Database Administrator at CTSI](#) (Memphis, TN). See this and other great job listings on [the jobs page](#).



Does that really matter? Three programmers, seven days, Waterloo Fortran IV. Are UML tools going to make or break it?

“I guess not.”

OK, so, what does it depend on?

“Will we have 19 inch monitors? And will we have access to all the Jolt we can drink?”

Again, does this matter? Is caffeine going to determine whether you can do it?

“I guess not. Oh, wait. You said I have a staff of two programmers?”

Right.

“Who are they?”

Does that matter?

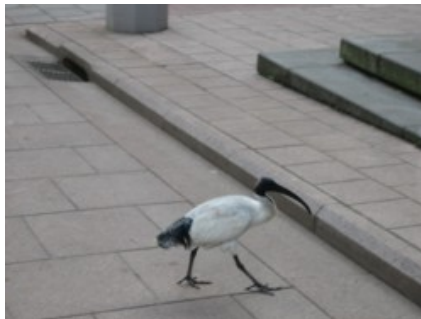
“Sure! If the team doesn’t get along, we’ll never be able to work together. And I know a few superstar programmers who could crank out a Fortran compiler *by themselves* in one week, and *lots of* programmers who couldn’t write the code to print the startup banner if they had six months.”

Now we’re on to something!

Everybody gives lip service to the idea that people are the most important part of a software project, but nobody is quite sure what you can *do* about it. The very first thing you have to do right if you want to have good programmers is to *hire* the right programmers, and that means you have to be able to figure out who the right programmers *are*, and this is usually done in the interview process. So this chapter is all about interviewing.

(The in-person interview is just one part of the hiring process, which starts with [sorting resumes](#) and a [phone screen](#). This article only covers in-person interviews.)

You should always try to have at least six people interview each candidate that gets hired, including at least five who would be peers of that candidate (that is, other programmers, not managers). You know the kind of company that just has some salty old manager interview each candidate, and that decision is the



only one that matters? These companies don’t have very good people working there. It’s too easy to fake out one interview, especially when a non-programmer interviews a programmer.

If even two of the six interviewers thinks that a person is not worth

hiring, don't hire them. That means you can technically end the "day" of interviews after the first two if the candidate is not going to be hired, which is not a bad idea, but to avoid cruelty you may not want to tell the candidate in advance how many people will be interviewing them. I have heard of companies that allow any interviewer to reject a candidate. This strikes me as a little bit too aggressive; I would probably allow any senior person to reject a candidate but would not reject someone just because one junior person didn't like them.

Don't try to interview a bunch of people at the same time. It's just not fair. Each interview should consist of one interviewer and one interviewee, in a room with a door that closes and a whiteboard. I can tell you from extensive experience that if you spend less than one hour on an interview you're not going to be able to make a decision.

You're going to see three types of people in your interviews. At one end of the scale, there are the unwashed masses, lacking even the most basic skills for this job. They are easy to ferret out and eliminate, often just by asking two or three quick questions. At the other extreme you've got your brilliant superstars who write lisp compilers for fun, in a weekend, in Assembler for the Nintendo DS. And in the middle, you have a large number of "maybes" who seem like they might just be able to contribute something. The trick is telling the difference between the superstars and the maybes, because the secret is that you don't want to hire any of the maybes. Ever.

At the end of the interview, you must be prepared to make a sharp decision about the candidate. There are only two possible outcomes to this decision: *Hire* or *No Hire*. There is no other possible answer. *Never* say, "Hire, but not for my team." This is rude and implies that the candidate is not smart enough to work with you, but maybe he's smart enough for those losers over in that other team. If you find yourself tempted to say "Hire, but not in my team," simply translate that mechanically to "No Hire" and you'll be OK. Even if you have a candidate that would be brilliant at doing your particular task, but wouldn't be very good in another team, that's a *No Hire*. In software, things change so often and so rapidly that you need people that can succeed at just about any programming task that you throw at them. If for some reason you find an idiot savant that is really, really, really good at SQL but completely incapable of ever learning any other topic, *No Hire*. You'll solve some short term pain in exchange for a lot of long term pain.

Never say "Maybe, I can't tell." If you can't tell, that means *No Hire*. It's really easier than you'd think. Can't tell? Just say no! If you are on the fence, that means *No Hire*. *Never* say, "Well, Hire, I guess, but I'm a little bit concerned about..." That's a *No Hire* as well. Mechanically translate all the waffling to "no" and you'll be all right.

Why am I so hardnosed about this? It's because it is much, *much* better to reject a good candidate than to accept a bad candidate. A bad candidate will cost a lot of money and effort and



waste other people's time fixing all their bugs. Firing someone you hired by mistake can take months and be nightmarishly difficult,



especially if they decide to be litigious about it. In some situations it may be completely impossible to fire anyone. Bad employees demoralize the good employees. And they might be bad programmers but really *nice people* or maybe they *really need this job*, so you can't bear to fire them, or you can't fire them without pissing everybody off, or whatever. It's just a bad scene.

On the other hand, if you reject a good candidate, I mean, I *guess* in some existential sense an injustice has been done, but, hey, if they're so smart, don't worry, they'll get *lots* of good job offers. Don't be afraid that you're going to reject too many people and you won't be able to find anyone to hire. During the interview, it's not your problem. Of course, it's important to seek out good candidates. But once you're actually interviewing someone, pretend that you've got 900 more people lined up outside the door. Don't lower your standards no matter how hard it seems to find those great candidates.

OK, I didn't tell you the most important part—how do you know whether to hire someone?

In principle, it's simple. You're looking for people who are

1. Smart, and
2. Get things done.

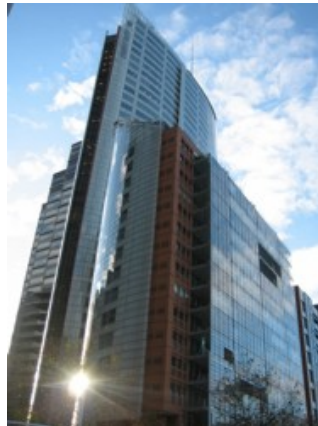
That's it. That's all you're looking for. Memorize that. Recite it to yourself before you go to bed every night. You don't have enough time to figure out much more in a short interview, so don't waste time trying to figure out whether the candidate might be pleasant to be stuck in an airport with, or whether they really know ATL and COM programming or if they're just faking it.

People who are *Smart* but don't *Get Things Done* often have PhDs and work in big companies where nobody listens to them because they are completely impractical. They would rather mull over something academic about a problem rather than ship on time. These kind of people can be identified because they love to point out the theoretical similarity between two widely divergent concepts. For example, they will say, "Spreadsheets are really just a special case of programming language," and then go off for a week and write a thrilling, brilliant whitepaper about the theoretical computational linguistic attributes of a spreadsheet as a programming language. Smart, but not useful. The other way to identify these people is that they have a tendency to show up at your office, coffee mug in hand, and try to start a long conversation about the relative merits of Java introspection vs. COM type libraries, *on the day you are trying to ship a beta*.

People who *Get Things Done* but are not *Smart* will do stupid things, seemingly without thinking about them, and somebody else will have

to come clean up their mess later. This makes them net *liabilities* to the company because not only do they fail to contribute, but they soak up good people's time. They are the kind of people who decide to refactor your core algorithms to use the Visitor Pattern, which they just read about the night before, and completely misunderstood, and instead of simple loops adding up items in an array you've got an `AdderVistior` class (yes, it's spelled wrong) and a `VisitationArrangingOfficer` singleton and none of your code works any more.

How do you detect *smart* in an interview? The first good sign is that you don't have to explain things over and over again. The conversation just flows. Often, the candidate says something that shows real insight, or brains, or mental acuity. So an important part of the interview is creating a situation where someone can show you how smart they are. The worst kind of interviewer is the blowhard. That's the kind who blabs the whole time and barely leaves the candidate time to say, "yes, that's *so* true, I *couldn't agree with you more*." Blowhards hire everyone; they think that the candidate must be smart because "he thinks so much like me!"



The second worst kind of interviewer is the Quiz Show Interviewer. This is the kind of person who thinks that smart means "knows a lot of facts." They just ask a bunch of trivia questions about programming and give points for correct answers. Just for fun, here is the worst interview question on Earth: "What's the difference between `varchar` and `varchar2` in Oracle 8i?" This is a terrible question. There is no possible, imaginable correlation between people that know that particular piece of trivia and people that you want to hire. Who cares what the difference is? You can find out online in about fifteen seconds! Remember, smart does *not* mean "knows the answer to trivia questions." Anyway, software teams want to hire people with *aptitude*, not a particular skill set. Any skill set that people can bring to the job will be technologically obsolete in a couple of years, anyway, so it's better to hire people that are going to be able to learn any new technology rather than people who happen to know how to make JDBC talk to a MySQL database *right this minute*.

But in general, the way to learn the most about a person is to let them do the talking. Give them open-ended questions and problems.

So, what do you ask?

My personal list of interview questions originates from my first job at Microsoft. There are actually hundreds of famous Microsoft interview questions. Everybody has a set of questions that they really like. You, too, will develop a particular set of questions and a personal interviewing style which helps you make the *Hire/No Hire* decision. Here are some techniques that I have used that have been successful.

Before the interview, I read over the candidates resume and jot down an interview plan on a scrap of paper. That's just a list of questions that I want to ask. Here's a typical plan for interviewing a programmer:

1. Introduction
2. Question about recent project candidate worked on
3. Easy Programming Question
4. Pointer/Recursion Question
5. Are you satisfied?
6. Do you have any questions?

I am very, very careful to avoid anything that might give me some preconceived notions about the candidate. If you think that someone is smart before they even walk into the room, just because they have a Ph.D. from MIT, then nothing they can say in one hour is going to overcome that initial prejudice. If you think they are a bozo because they went to community college, nothing they can say will overcome that initial impression. An interview is like a very, very delicate scale—it's very hard to judge someone based on a one hour interview and it may seem like a very close call. But if you know a little bit about the candidate beforehand, it's like a big weight on one side of the scale, and the interview is useless. Once, right before an interview, a recruiter came into my office. "You're going to *love* this guy," she said. *Boy* did this make me mad. What I should have said was, "Well, if you're so sure I'm going to love him, why don't you just hire him instead of wasting my time going through this interview." But I was young and naïve, so I interviewed him. When he said not-so-smart things, I thought to myself, "gee, must be the exception that proves the rule." I looked at everything he said through rose-colored glasses. I wound up saying *Hire* even though he was a crappy candidate. You know what? Everybody else who interviewed him said *No Hire*. So: don't listen to recruiters; don't ask around about the person before you interview them; and never, ever talk to the other interviewers about the candidate until you've both made your decisions independently. That's the scientific method.

The *introduction* phase of the interview is intended to put the candidate at ease. I ask them if they had a nice flight. I spend about 30 seconds telling the person who I am and how the interview will work. I always reassure candidates that we are interested in *how* they go about solving problems, not the actual answer.



Part two is a question about some recent project that the candidate worked on. For interviewing college kids, ask them about their senior thesis, if they had one, or about a course they took that involved a long project that they really enjoyed. For example, sometimes I will ask, "what class did you take last semester that you liked the most? It

doesn't have to be computer-related." When interviewing experienced candidates, you can talk about their most recent assignment from their previous job.

Again, ask open-ended questions and sit back and listen, with only the occasional "tell me more about that" if they seem to stall.

What should you look for during the open ended questions?

One: Look for passion. Smart people are passionate about the projects they work on. They get very excited talking about the subject. They talk quickly, and get animated. Being passionately *negative* can be just as good a sign. "My last boss wanted to do everything on VAX computers because it was all he understood. What a dope!" There are far too many people around that can work on something and not really care one way or the other. It's hard to get people like this motivated about anything.

Bad candidates just don't care and will not get enthusiastic at all during the interview. A really good sign that a candidate is passionate about something is that when they are talking about it, they will forget for a moment that they are in an interview. Sometimes a candidate comes in who is very nervous about being in an interview situation—this is normal, of course, and I always ignore it. But then when you get them talking about Computational Monochromatic Art they will get extremely excited and lose all signs of nervousness. Good. I like passionate people who really care. (To see an example of Computational Monochromatic Art try unplugging your monitor.) You can challenge them on something (try it—wait for them to say something that's probably true and say "that couldn't be true") and they will defend themselves, even if they were sweating five minutes ago, because they care so much they forget that you are going to be making Major Decisions About Their Life soon.

Two: Good candidates are careful to explain things well, at whatever level. I have rejected candidates because when they talked about their previous project, they couldn't explain it in terms that a normal person could understand. Often CS majors will just assume that everyone knows what Bates Theorem is or what $O(\log n)$ means. If they start doing this, stop them for a minute and say, "could you do me a favor, just for the sake of the exercise, could you please explain this in terms my grandmother could understand." At this point many people will *still* continue to use jargon and will completely fail to make themselves understood. *Gong!* You don't want to hire them, basically, because they are not smart enough to comprehend what it takes to make other people understand their ideas.

Three: If the project was a team project, look for signs that they took a leadership role. A candidate might say, "We were working on X, but the boss said Y and the client said Z." I'll ask, "So what did *you* do?" A good answer to this might be "I got together with the other members of the team and wrote a proposal..." A bad answer might be, "Well, there was nothing I *could* do. It was an impossible situation." Remember, *Smart* and *Gets Things Done*. The only way you're going to be able to tell if somebody *Gets Things Done* is to see if historically

they have tended to get things done in the past. In fact, you can even ask them directly to give you an example from their recent past when they took a leadership role and got something done—overcoming some institutional inertia, for example.

Most of the time in the interview, though, should be spent letting the candidate prove that they can write code.

Reassure candidates that you understand that it's hard to write code without an editor, and you will forgive them if the whiteboard gets really messy. Also you understand that it's hard to write bug-free code without a compiler, and you will take that into account.



For the first interview of the day, I've started including a really, really easy programming problem. I had to start doing this during the dotcom boom when a lot of people who thought HTML was "programming" started showing up for interviews, and I needed a way to avoid wasting too much time with them. It's the kind of problem that any programmer working today should be able to solve in about one minute. Some examples:

1. Write a function that determines if a string starts with an upper-case letter A-Z
2. Write a function that determines the area of a circle given the radius
3. Add up all the values in an array

These softball questions seem too easy, so when I first started asking them, I had to admit that I really expected everyone to sail right through them. What I discovered was that everybody *solved* the problem, but there was a lot of variation in *how long* it took them to solve.

That reminded me of why I couldn't trade bonds for a living.

Jared is a bond trader. He is always telling me about interesting deals that he did. There's this thing called an option, and there are puts, and calls, and the market steepens, so you put on steepeners, and it's all very confusing, but the weird thing is that *I know what all the words mean*, I know exactly what a put is (the right, but not the responsibility, to sell something at a certain price) and in only three minutes I can figure out what should happen if you own a put and the market goes up, but I need the *full* three minutes to figure it out, and when he's telling me a more complicated story, where the puts are just the first bit, there are lots of other bits to the story, I lose track very quickly, because I'm lost in thought ("let's see, market goes up, that mean interest rates go *down*, and now, a put is the right to sell something...") until he gets out the graph paper and starts walking me through it, and my eyes glazeth over and it's very sad. Even though I understand all the little bits, I can't understand them *fast enough* to

get the big picture.

And the same thing happens in programming. If the basic concepts aren't so easy that you don't even have to think about them, you're not going to get the big concepts.

Serge Lang, a math professor at Yale, used to give his Calculus students a fairly simple algebra problem on the first day of classes, one which almost everyone could solve, but some of them solved it *as quickly as they could write* while others took a while, and Professor Lang claimed that all of the students who solved the problem as quickly as they could write would get an A in the Calculus course, and all the others wouldn't. The *speed* with which they solved a simple algebra problem was as good a predictor of the final grade in Calculus as a whole semester of homework, tests, midterms, and a final.

You see, if you can't whiz through the *easy* stuff at 100 m.p.h., you're never gonna get the advanced stuff.

But like I said, the good programmers stand up, write the answer on the board, sometimes adding a clever fillip (Ooh! Unicode compliant! Nice!), and it takes thirty seconds, and now I have to decide if they're really good, so I bring out the big guns: recursion and pointers.

15 years of experience interviewing programmers has convinced me that the best programmers all have an easy aptitude for dealing with multiple levels of abstraction simultaneously. In programming, that means specifically that they have no problem with recursion (which involves holding in your head multiple levels of the call stack at the same time) or complex pointer-based algorithms (where the address of an object is sort of like an abstract representation of the object itself).



I've come to realize that understanding pointers in C is not a skill, it's an aptitude. In first year computer science classes, there are always about 200 kids at the beginning of the semester, all of whom wrote complex adventure games in BASIC for their PCs when they were 4 years old. They are having a good ol' time learning C or Pascal in college, until one day the professor introduces pointers, and suddenly, *they don't get it*. They just don't understand anything any more. 90% of the class goes off and becomes Political Science majors, then they tell their friends that there weren't enough good looking members of the appropriate sex in their CompSci classes, that's why they switched. *For some reason most people seem to be born without the part of the brain that understands pointers*. Pointers require a complex form of doubly-indirected thinking that some people just can't do, and it's pretty crucial to good programming. A lot of the "script jocks" who started programming by copying JavaScript snippets into their web pages and went on to learn Perl never learned about pointers, and they can never quite produce code of the quality

you need.

That's the source of all these famous interview questions you hear about, like "reversing a linked list" or "detect loops in a tree structure."

Sadly, despite the fact that I think that all good programmers should be able to handle recursion and pointers, and that this is an excellent way to tell if someone is a good programmer, the truth is that these days, programming languages have almost completely made that specific art unnecessary. Whereas ten years ago it was rare for a computer science student to get through college without learning recursion and functional programming in one class and C or Pascal with data structures in another class, today [it's possible in many otherwise reputable schools to coast by on Java alone](#).

A lot of programmers that you might interview these days are apt to consider recursion, pointers, and even data structures to be a silly implementation detail which has been abstracted away by today's many happy programming languages. "When was the last time you had to write a sorting algorithm?" they snicker.

Still, I don't really care. I want my ER doctor to understand anatomy, even if all she has to do is put the computerized defibrillator nodes on my chest and push the big red button, and I want programmers to know programming down to the CPU level, even if Ruby on Rails *does* read your mind and build a complete Web 2.0 social collaborative networking site for you with three clicks of the mouse.

Even though the format of the interview is, superficially, just a candidate writing some code on the whiteboard, my real goal here is to have a conversation about it. "Why did you do it that way?" "What are the performance characteristics of your algorithm?" "What did you forget?" "Where's your bug?"



That means I don't really mind giving programming problems that are too hard, as long as the candidate has some chance of starting out, and then I'm happy to dole out little hints along the way, little footholds, so to speak. I might ask someone, say, to project a triangle onto a plane, a typical graphics problem, and I don't mind helping them with the trig (SOH-CAH-TOA, baby!), and when I ask them how to speed it up, I might drop little hints about look-up tables. Notice that the kinds of hints I'm happy to provide are really just answers to trivia questions—the kinds of things that you find on Google.

Inevitably, you will see a bug in their function. So we come to question five from my interview plan: "Are you satisfied with that code?" You may want to ask, "OK, so where's the bug?" The quintessential Open Ended Question From Hell. All programmers make mistakes, there's nothing wrong with that, they just have to be

able to find them. With string functions in C, most college kids forget to null-terminate the new string. With almost any function, they are likely to have off-by-one errors. They will forget semicolons sometimes. Their function won't work correctly on 0 length strings, or it will GPF if malloc fails... Very, very rarely, you will find a candidate that doesn't have any bugs the first time. In this case, this question is even more fun. When you say, "There's a bug in that code," they will review their code carefully, and then you get to see if they can be diplomatic yet firm in asserting that the code is perfect.

As the last step in an interview, ask the candidate if they have any questions. Remember, even though you're interviewing them, the good candidates have lots of choices about where to work and they're using this day to figure out if they want to work for you.

Some interviewees try to judge if the candidate asks "intelligent" questions. Personally, I don't care what questions they ask; by this point I've already made my decision. The trouble is, candidates have to see about five or six people in one day, and it's hard for them to ask five or six people different, brilliant questions, so if they don't have any questions, fine.

I always leave about five minutes at the end of the interview to sell the candidate on the company and the job. This is actually important *even if you are not going to hire them*. If you've been lucky enough to find a really good candidate, you want to do everything you can at this point to make sure that they want to come work for you. Even if they are a bad candidate, you want them to like your company and go away with a positive impression.

Ah, I just remembered that I should give you some more examples of really bad questions.

First of all, avoid the illegal questions. Anything related to race, religion, gender, national origin, age, military service eligibility, veteran status, sexual orientation, or physical handicap is illegal here in the United States. If their resume says they were in the Marines, you can't ask them, even to make pleasant conversation, if they were in Iraq. It's against the law to discriminate based on veteran status. If their resume says that they attended the Technion in Haifa, don't ask them, even conversationally, if they are Israeli, even if you're just making conversation because your wife is Israeli, or you love Felafel. It's against the law to discriminate based on national origin.

Next, avoid any questions which might make it seem like you care about, or are discriminating based on, things which you don't actually care about or discriminate based on. The best example of this I can think of is asking someone if they have kids or if they are married. This might give the impression that you think that people with kids aren't going to devote enough time to their work or that they are going to run off and take maternity leave. Basically, stick to questions which are completely relevant to the job for which they are interviewing.

Finally, avoid brain teaser questions like the one where you have to

arrange 6 equal length sticks to make exactly 4 identical perfect triangles. Or anything involving pirates, marbles, and secret codes. Most of these are “Aha!” questions—the kind of question where either you know the answer or you don’t. With these questions knowing the answer just means you heard that brain teaser before. So as an interviewer, you don’t get any information about “smart/get things done” by figuring out if they happen to make a particular mental leap.

In the past, I’ve used “impossible questions,” also known as “back of the envelope questions.” Classic examples of this are “How many piano tuners are there in Seattle?” The candidate won’t know the answer, but smart candidates won’t give up and they’ll be happy to try and estimate a reasonable number for you. Let’s see, there are probably... what, a million people in Seattle? And maybe 1% of them have pianos? And a piano needs to be tuned every couple of years? And it takes 35 minutes to tune one? All wrong, of course, but at least they’re attacking the problem. The only reason to ask a question like this is that it lets you have a conversation with the candidate. “OK, 35 minutes, but what about travel time between pianos?”

“Good point. If the piano tuner could take reservations well in advance they could probably set up their schedule to minimize travel time. You know, do all the pianos in Redmond on Monday rather than going back and forth across 520 three times a day.”

A good back-of-the-envelope question allows you to have a conversation with the candidate that helps you form an opinion about whether they are smart. A bad “Aha!” pirate question usually results in the candidate just sort of staring at you for a while and then saying they’re stuck.

If, at the end of the interview, you’ve convinced yourself that this person is *smart* and *gets things done*, and four or five other interviewers agree, you probably won’t go wrong in hiring them. But if you have any doubts, you’re better off waiting for someone better.

The optimal time to make a decision about the candidate is about three minutes after the end of the interview. Far too many companies allow interviewers to wait days or weeks before turning in their feedback. Unfortunately, the more time that passes, the less you’ll remember.

I ask interviewers to write *immediate* feedback after the interview, either a “hire” or “no hire”, followed by a one or two paragraph justification. It’s due 15 minutes after the interview ends.

If you’re having trouble deciding, there’s a very simple solution. NO HIRE. Just don’t hire people that you aren’t sure about. This is a little nerve wracking the first few times—what if we *never* find someone good? That’s OK. If your resume and phone-screening process is working, you’ll probably have about 20% hires in the live interview. And when you find the smart, gets-things-done candidate, *you’ll know it*. If you’re not thrilled with someone, move on.

Next: [Choices = Headaches](#)

Want to know more? You're reading [Joel on Software](#), stuffed with years and years of completely raving mad articles about software development, managing software teams, designing user interfaces, running successful software companies, and rubber duckies.

About the author. I'm [Joel Spolsky](#), co-founder of [Fog Creek Software](#), a New York company that proves that you can treat programmers well and still be highly profitable. Programmers get private offices, free lunch, and work 40 hours a week. Customers only pay for software if they're delighted. We make FogBugz, an enlightened [bug tracker](#) designed to help great teams develop brilliant software, Kiln, which simplifies source control and [code review](#), and Fog Creek Copilot, which makes [remote desktop control](#) easy. I'm also the co-founder of [Stack Overflow](#).

© 2000-2011 Joel Spolsky
joel@joelonsoftware.com

Have you been wondering about Distributed Version Control? It has been a huge productivity boon for us, so I wrote Hg Init, a [Mercurial tutorial](#)—check it out!

