Joel on Software

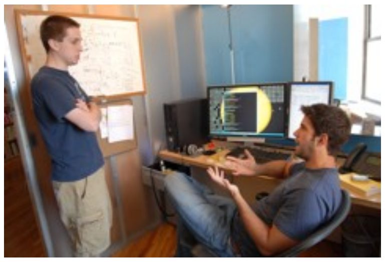
# Evidence Based Scheduling

*by Joel Spolsky*

Friday, October 26, 2007

Software developers don't really like to make schedules. Usually, they try to get away without one. "It'll be done when it's done!" they say, expecting that such a brave, funny zinger will reduce their boss to a fit of giggles, and in the ensuing joviality, the schedule will be forgotten.

Most of the schedules you do see are halfhearted attempts. They're stored on a file share somewhere and completely forgotten. When these teams ship, two years late, that weird guy with the file cabinet in his office brings the old printout to the post mortem, and everyone has a good laugh. "Hey look! We allowed two weeks for rewriting from scratch in Ruby!"

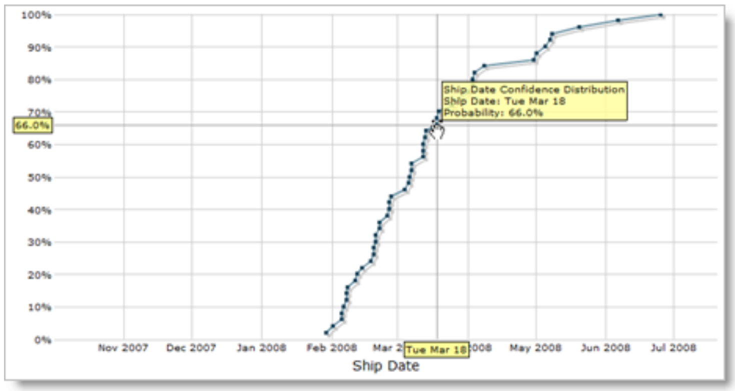


Hilarious! If you're still in business.

You want to be spending your time on things that get the most bang for the buck. And you can't figure out how much buck your bang is going to cost without knowing how long it's going to take. When you have to decide between the "animated paperclip" feature and the "more financial functions" feature, you really need to know how much time each will take.

Why won't developers make schedules? Two reasons. One: it's a pain in the butt. Two: nobody believes the schedule is realistic. Why go to all the trouble of working on a schedule if it's not going to be right?

Over the last year or so at Fog Creek we've been developing a system that's so easy even our grouchiest developers are willing to go along with it. And as far as we can tell, it produces extremely reliable schedules. It's called Evidence-Based Scheduling, or EBS. You gather *evidence*, mostly from historical timesheet data, that you feed back into your schedules. What you get is not just one ship date: you get a confidence distribution curve, showing the probability that you will ship on any given date. It looks like this:



The steeper the curve, the more confident you are that the ship date is real.

Here's how you do it.

## 1) Break 'er down

When I see a schedule measured in days, or even weeks, I know it's not going to work. You have to break your schedule into very small tasks that can be measured in *hours*. Nothing longer than 16 hours.

This forces you to actually figure out what you are going to do. Write subroutine *foo*. Create this dialog box. Parse the Fizzbott file. Individual development tasks are easy to estimate, because you've written subroutines, created dialogs, and parsed files before.

If you are sloppy, and pick big three-week tasks (e.g., "Implement Ajax photo editor"), then you *haven't thought about what you are going to do*. In detail. Step by step. And when you haven't thought about what you're going to do, you can't know how long it will take.

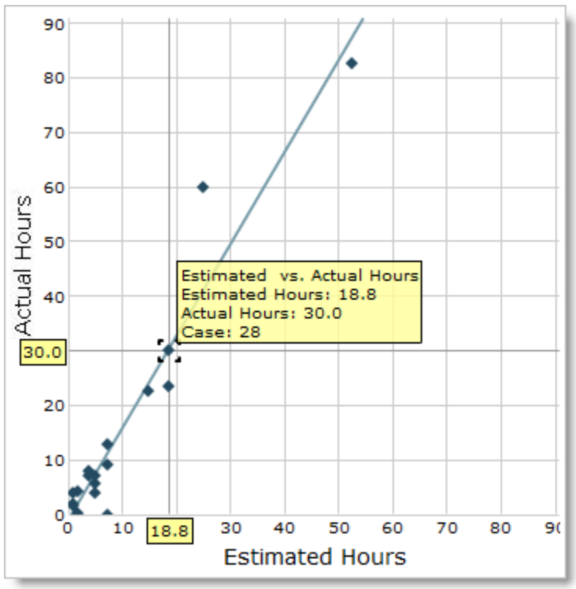
Setting a 16-hour maximum forces you to *design* the damn feature. If you have a hand-wavy three week feature called "Ajax photo editor" without a detailed design, I'm sorry to be the one to break it to you but you are officially *doomed*. You never thought about the steps it's going to take and you're sure to be forgetting a lot of them.

## 2) Track elapsed time

It's hard to get individual estimates exactly right. How do you account for interruptions, unpredictable bugs, status meetings, and the semiannual Windows Tithe Day when you have to reinstall everything from scratch on your main development box? Heck, even without all that stuff, how can you tell exactly how long it's going to take to implement a given subroutine?

You can't, really.

So, keep timesheets. Keep track of how long you spend working on each task. Then you can go back and see how long things took relative to the estimate. For each developer, you'll be collecting data like this:



Each point on the chart is one completed task, with the estimate and actual times for that task. When you divide estimate by actual, you get *velocity*: how fast the task was done relative to estimate. Over time, for each developer, you'll collect a history of velocities.

- The mythical *perfect estimator*, who exists only in your imagination, always gets every estimate exactly right. So their velocity history is {1, 1, 1, 1, 1, ...}
- A typical *bad estimator* has velocities all over the map, for example {0.1, 0.5, 1.7, 0.2, 1.2, 0.9, 13.0}
- Most estimators get the scale wrong but the relative estimates right. Everything takes longer than expected, because the estimate didn't account for bug fixing, committee meetings, coffee breaks, and that crazy boss who interrupts all the time. This *common estimator* has very consistent velocities, but they're below 1.0. For example, {0.6, 0.5, 0.6, 0.6, 0.5, 0.6, 0.7, 0.6}

As estimators gain more experience, their estimating skills improve. So throw away any velocities older than, say, six months.

If you have a new estimator on your team, who doesn't have a track record, assume the worst: give them a fake history with a wide range of velocities, until they've finished a half-dozen real tasks.

## 3) Simulate the future

Rather than just adding up estimates to get a single ship date, which sounds right but gives you a profoundly wrong result, you're going to use the Monte Carlo method to simulate many possible futures. In a Monte Carlo simulation, you can create 100 possible scenarios for the future. Each of these possible futures has 1% probability, so you can make a chart of the probability that you will ship by any given date.

While calculating each possible future for a given developer, you're going divide each task's estimate by a *randomly-selected velocity* from that developer's historical velocities, which we've been gathering in step 2. Here's one sample future:

| | | | | | |
|---|---|---|---|---|---|
| Estimate: | 4 | 8 | 2 | 8 | 16 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Random Velocity: | 0.6 | 0.5 | 0.6 | 0.6 | 0.5 | **Total:** |
| E/V: | **6.7** | **16** | **3.3** | **13.3** | **32** | **71.3** |

Do that 100 times; each total has 1% probability, and now you can figure out the probability that you will ship on any given date.

Now watch what happens:

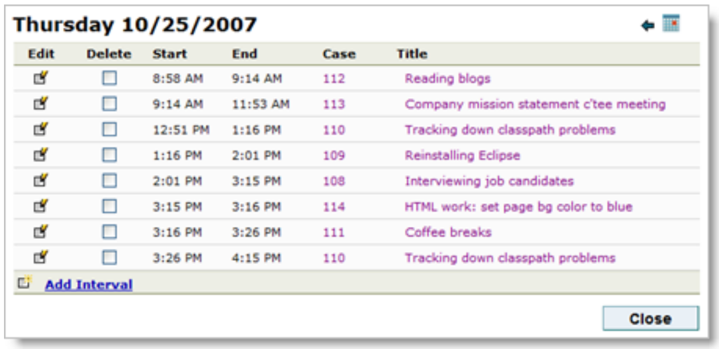
- In the case of the mythical perfect estimator, all velocities are 1. Dividing by a velocity which is always 1 has no effect. Thus, all rounds of the simulation give the same ship date, and that ship date has 100% probability. Just like in the fairy tales!
- The bad estimator's velocities are all over the map. 0.1 and 13.0 are just as likely. Each round of the simulation is going to produce a very different result, because when you divide by random velocities you get very different numbers each time. The probability distribution curve you get will be very shallow, showing an equal chance of shipping tomorrow or in the far future. That's still useful information to get, by the way: it tells you that you shouldn't have confidence in the predicted ship dates.
- The common estimator has a lot of velocities that are pretty close to each other, for example, {0.6, 0.5, 0.6, 0.6, 0.5, 0.6, 0.7, 0.6}. When you divide by these velocities you increase the amount of time something takes, so in one iteration, an 8-hour task might 13 hours; in another it might take 15 hours. That compensates for the estimators perpetual optimism. And it compensates *precisely*, based *exactly* on this developers *actual, proven, historical optimism*. And since all the historical velocities are pretty close, hovering around 0.6, when you run each round of the simulation, you'll get pretty similar numbers, so you'll wind up with a narrow range of possible ship dates.

In each round of the Monte Carlo simulation, of course, you have to convert the hourly data to calendar data, which means you have to take into account each developer's work schedule, vacations, holidays, etc. And then you have to see, for each round, which developer is finishing last, because that's when the whole team will be done. These calculations are painstaking, but luckily, painstaking is what computers are good at.

## Obsessive-compulsive disorder not required

What do you do about the boss who interrupts you all the time with long-winded stories about his fishing trips? Or the sales meetings you're forced to go to even though you have no reason to be there? Coffee breaks? Spending half a day helping the new guy get his dev environment set up?

When Brett and I were developing this technique at Fog Creek, we worried a lot about things that take real time but can't be predicted in advance. Sometimes, this all adds up to more time than writing code. Should you have estimates for this stuff too, and track it on a time sheet?



Thursday 10/25/2007

| Edit | Delete | Start | End | Case | Title |
|---|---|---|---|---|---|
| | ☐ | 8:58 AM | 9:14 AM | 112 | Reading blogs |
| | ☐ | 9:14 AM | 11:53 AM | 113 | Company mission statement c'tee meeting |
| | ☐ | 12:51 PM | 1:16 PM | 110 | Tracking down classpath problems |
| | ☐ | 1:16 PM | 2:01 PM | 109 | Reinstalling Eclipse |
| | ☐ | 2:01 PM | 3:15 PM | 108 | Interviewing job candidates |
| | ☐ | 3:15 PM | 3:16 PM | 114 | HTML work: set page bg color to blue |
| | ☐ | 3:16 PM | 3:26 PM | 111 | Coffee breaks |
| | ☐ | 3:26 PM | 4:15 PM | 110 | Tracking down classpath problems |

Well, yeah, you can, if you want. And Evidence Based Scheduling will work.

*But you don't have to.*

It turns out that EBS works so well that all you have to do is *keep the clock running* on whatever task you were doing when the interruption occurred. As disconcerting as this may sound, EBS produces the best results when you do this.

Let me walk you through a quick example. To make this example as simple as possible, I'm going to imagine a very predictable programmer, John, whose whole job is writing those one-line getter and setter functions that inferior programming languages require. All day long this is all he does:

```
private int width;
public int getWidth () { return width; }
public void setWidth (int _width} { width =
_width; }
```

I know, I know… it's a deliberately dumb example, but you *know* you've met someone like this.

Anyway. Each getter or setter takes him 2 hours. So his task estimates look like this:

{2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, … }

Now, this poor guy has a boss who interrupts him every once in a while with a two-hour conversation about marlin fishing. Now, of course, John could have a task on his schedule called "Painful conversations about marlin," and put that on his timesheet, but this might not be politically prudent. Instead, John just keeps the clock running. So his actual times look like this:

{2, 2, 2, 2, 4, 2, 2, 2, 2, 4, 2, … }

And his velocities are:

{1, 1, 1, 1, 0.5, 1, 1, 1, 1, 0.5, 1, … }

Now think about what happens. In the Monte Carlo simulation, the probability that each estimate will be divided by 0.5 *is exactly the same as the probability that John's boss would interrupt him during* *any given feature*. So EBS produces a correct schedule!

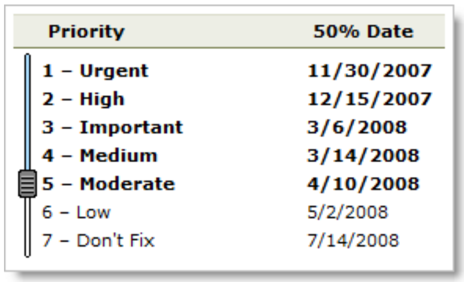
In fact, EBS is far more likely to have accurate evidence about these interruptions than even the most timesheet-obsessive developer. *Which is exactly why it works so well*. Here's how I explain this to people. When developers get interrupted, they can either

1. make a big stink about putting the interruption on their timesheet and in their estimates, so management can see just how much time is being wasted on fishing conversation, or
2. make a big stink about refusing to put it on their timesheet, just letting the feature they were working on slip, because they refuse to pad *their* estimates which were *perfectly correct* with stupid conversation about fishing expeditions to which they *weren't even invited*,

... and in either case, *EBS gives the same, exactly correct results*, no matter which type of passive-aggressive developer you have.

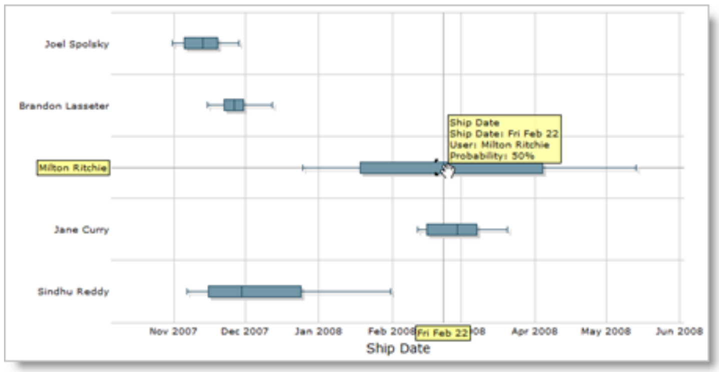
## 4) Manage your projects actively

Once you've got this set up, you can actively manage projects to ship on time. For example, if you sort features out into different priorities, it's easy to see how much it would help the schedule if you could cut the lower priority features.



| Priority | 50% Date |
|---|---|
| **1 – Urgent** | **11/30/2007** |
| **2 – High** | **12/15/2007** |
| **3 – Important** | **3/6/2008** |
| **4 – Medium** | **3/14/2008** |
| **5 – Moderate** | **4/10/2008** |
| 6 – Low | 5/2/2008 |
| 7 – Don't Fix | 7/14/2008 |

You can also look at the distribution of possible ship dates for *each developer:*



Some developers (like Milton in this picture) may be causing problems because their ship dates are so uncertain: they need to work on learning to estimate better. Other developers (like Jane) have very precise ship dates that are just too late: they need to have some of their work taken off their plate. Other developers (me! yay!) are not on the critical path at all, and can be left in peace.

## Scope creep

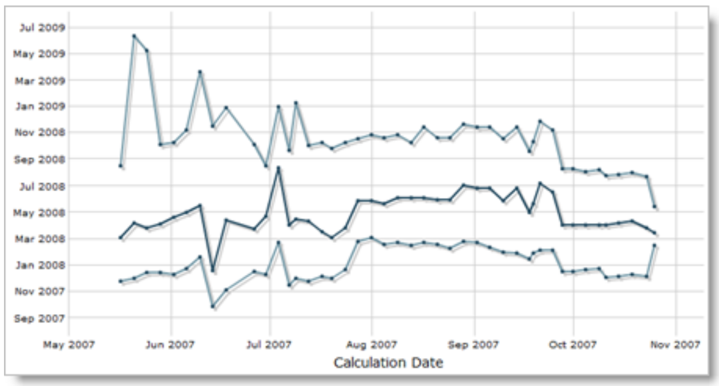
Assuming you had everything planned down to the last detail when you started work, EBS works great. To be honest, though, you may do some features that you hadn't planned. You get new ideas, your salespeople sell features you don't have, and somebody on the board of directors comes up with a cool new idea to make your golf cart GPS application monitor EKGs while golfers are buzzing around the golf course. All this leads to delays that could not have been predicted when you did the original schedule.

Ideally, you have a bunch of buffer for this. In fact, go ahead and build buffer into your original schedule for:

1. New feature ideas
2. Responding to the competition
3. Integration (getting everyone's code to work together when it's merged)
4. Debugging time
5. Usability testing (and incorporating the results of those tests into the product).
6. Beta tests

So now, when new features come up, you can slice off a piece of the appropriate buffer and use it for the new feature.

What happens if you're still adding features and you've run out of buffer? Well, now the ship dates you get out of EBS start slipping. You should take a snapshot of the ship date confidence distribution every night, so that you can track this over time:



The *x*-axis is when the calculation was done; the *y*-axis is the ship date. There are three curves here: the top one is the 95% probability date, the middle is 50% and the bottom is 5%. So, the closer the curves are to one another, the narrower the range of possible ship dates.

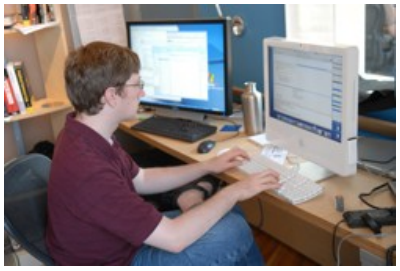
If you see ship date getting later and later (rising curves), you're in trouble. If it's getting later by more than one day per day, you're adding work faster than you're completing work, and you'll never be done. You can also look and see if the ship date confidence distribution is getting tighter (the curves are converging), which it should be if you're really converging on a date.

## While we're at it

Here are a few more things I've learned over the years about schedules.

**1) Only the programmer doing the work can create the estimate.** Any system where management writes a schedule and hands it off to programmers is doomed to fail. Only the programmer who is going to implement a feature can figure out what steps they will need to take to implement that feature.

**2) Fix bugs as you find them, and charge the time back to the original task.** You can't schedule a single bug fix in advance, because you don't know what bugs you're going to have. When bugs are found in new code, charge the time to the original task that you implemented incorrectly. This will help EBS predict the time it takes to get *fully debugged* code, not just *working* code.



**3) Don't let managers badger developers into shorter estimates.** Many rookie software managers think that they can "motivate" their programmers to work faster by giving them nice, "tight" (unrealistically short) schedules. I think this kind of motivation is brain-dead. When I'm behind schedule, I feel doomed and depressed and unmotivated. When I'm working *ahead* of schedule, I'm cheerful and productive. The schedule is not the place to play psychological games.

Why do managers try this?

When the project begins, the technical managers go off, meet with the business people, and come up with a list of features they *think* would take about three months, but which would really take twelve. When you think of writing code without thinking about all the steps you have to take, it always seems like it will take *n* time, when in reality it will probably take more like 4*n* time. When you do a real schedule, you add up all the tasks and realize that the project is going to take much longer than originally thought. The business people are unhappy.

Inept managers try to address this by figuring out how to get people to work faster. This is not very realistic. You might be able to hire more people, but they need to get up to speed and will probably be working at 50% efficiency for several months (and dragging down the efficiency of the people who have to mentor them).

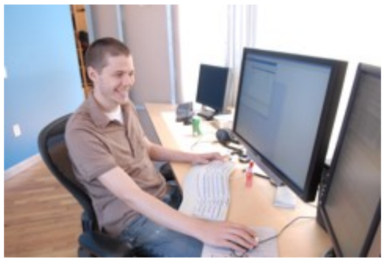
You might be able to get 10% more raw code out of people *temporarily* at the cost of having them burn out 100% in a year. Not a big gain, and it's a bit like eating your seed corn. Of course, when you overwork people, debugging time *doubles* and a late project becomes later. Splendid karma.

But you can never get 4*n* from *n*, ever, and if you think you can, please email me the stock symbol for your company so I can short it.

**4) A schedule is a box of wood blocks.** If you have a bunch of wood blocks, and you can't fit them into a box, you have two choices: get a bigger box, or remove some blocks. If you wanted to ship in six months, but you have twelve months on the schedule, you are either going to have to delay shipping, or find some features to delete. You just can't shrink the blocks, and if you pretend you can, then you are merely depriving yourself of a useful opportunity to actually *see into the future* by lying to yourself about what you see there.

Now that I mention it, one of the great benefits of realistic schedules is that you *are* forced to delete features. Why is this good?

Suppose you have two features in mind. One is really useful and will make your product really great. The other is really easy and the programmers can't wait to code it up ("Look! <blink>!"), but it serves no useful purpose.



If you don't make a schedule, the programmers will do the easy/fun feature first. Then they'll run out of time, and you will have no choice but to slip the schedule to do the useful/important feature.

If you do make a schedule, even before you start working, you'll realize that you have to cut something, so you'll cut the easy/fun feature and just do the useful/important feature. By forcing yourself to chose some features to cut, you wind up making a more powerful, better product with a better mix of good features that ships sooner.

Way back when I was working on Excel 5, our initial feature list was huge and would have gone *way* over schedule. "Oh my!" we thought. "Those are *all* super important features! How can we live without a macro editing wizard?"

As it turns out, we had no choice, and we cut what we thought was "to the bone" to make the schedule. Everybody felt unhappy about the cuts. To make people feel better, we told ourselves that we weren't *cutting* the features, we were simply *deferring them* to Excel 6.

As Excel 5 was nearing completion, I started working on the Excel 6 spec with a colleague, Eric Michelman. We sat down to go through the list of "Excel 6" features that had been punted from the Excel 5 schedule. Guess what? It was the shoddiest list of features you could imagine. Not *one* of those features was worth doing. I don't think a single one of them ever was. The process of culling features to fit a schedule was the best thing we could have done. If we hadn't done this, Excel 5 would have taken twice as long and included 50% useless crap features that would have had to be supported, for backwards compatibility, until the end of time.

## Summary

Using Evidence-Based Scheduling is pretty easy: it will take you a day or two at the beginning of every iteration to produce detailed estimates, and it'll take a few seconds every day to record when you start working on a new task on a timesheet. The benefits, though, are huge: realistic schedules.

Realistic schedules are the key to creating good software. It forces you to do the best features first and allows you to make the right decisions about what to build. Which makes your product better, your boss happier, delights your customers, and—best of all—lets you go home at five o'clock.

## P.S.

Evidence Based Scheduling is built into FogBugz 6.0.

**Next:** How to demo software

**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, 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 project management system designed to help great teams develop brilliant software, and Fog Creek Copilot, which makes remote desktop access easy.