Holding a Program in One's Head

paulgraham.com · Paul Graham · 2007-08 · source

|
August 2007

A good programmer working intensively on his own code can hold it in his mind the way a mathematician holds a problem he's working on. Mathematicians don't answer questions by working them out on paper the way schoolchildren are taught to. They do more in their heads: they try to understand a problem space well enough that they can walk around it the way you can walk around the memory of the house you grew up in. At its best programming is the same. You hold the whole program in your head, and you can manipulate it at will.

That's particularly valuable at the start of a project, because initially the most important thing is to be able to change what you're doing. Not just to solve the problem in a different way, but to change the problem you're solving.

Your code is your understanding of the problem you're exploring. So it's only when you have your code in your head that you really understand the problem.

It's not easy to get a program into your head. If you leave a project for a few months, it can take days to really understand it again when you return to it. Even when you're actively working on a program it can take half an hour to load into your head when you start work each day. And that's in the best case. Ordinary programmers working in typical office conditions never enter this mode. Or to put it more dramatically, ordinary programmers working in typical office conditions never really understand the problems they're solving.

Even the best programmers don't always have the whole program they're working on loaded into their heads. But there are things you can

do to help:

It's striking how often programmers manage to hit all eight points by accident. Someone has an idea for a new project, but because it's not officially sanctioned, he has to do it in off hours—which turn out to be more productive because there are no distractions. Driven by his enthusiasm for the new project he works on it for many hours at a stretch. Because it's initially just an experiment, instead of a "production" language he uses a mere "scripting" language—which is in fact far more powerful. He completely rewrites the program several times; that wouldn't be justifiable for an official project, but this is a labor of love and he wants it to be perfect. And since no one is going to see it except him, he omits any comments except the note-to-self variety. He works in a small group perforce, because he either hasn't told anyone else about the idea yet, or it seems so unpromising that no one else is allowed to work on it. Even if there is a group, they couldn't have multiple people editing the same code, because it changes too fast for that to be possible. And the project starts small because the idea is small at first; he just has some cool hack he wants to try out. Avoid distractions. Distractions are bad for many types of work, but especially bad for programming, because programmers tend to operate at the limit of the detail they can handle.

The danger of a distraction depends not on how long it is, but on how much it scrambles your brain. A programmer can leave the office and go and get a sandwich without losing the code in his head. But the wrong kind of interruption can wipe your brain in 30 seconds.

Oddly enough, scheduled distractions may be worse than unscheduled ones. If you know you have a meeting in an hour, you don't even start working on something hard.

Work in long stretches. Since there's a fixed cost each time you start working on a program, it's more efficient to work in a few long sessions than many short ones. There will of course come a point where you get stupid because you're tired. This

varies from person to person. I've heard of people hacking for 36 hours straight, but the most I've ever been able to manage is about 18, and I work best in chunks of no more than 12.

The optimum is not the limit you can physically endure. There's an advantage as well as a cost of breaking up a project. Sometimes when you return to a problem after a rest, you find your unconscious mind has left an answer waiting for you.

Use succinct languages. More powerful programming languages make programs shorter. And programmers seem to think of programs at least partially in the language they're using to write them. The more succinct the language, the shorter the program, and the easier it is to load and keep in your head.

You can magnify the effect of a powerful language by using a style called bottom-up programming, where you write programs in multiple layers, the lower ones acting as programming languages for those above. If you do this right, you only have to keep the topmost layer in your head.

Keep rewriting your program. Rewriting a program often yields a cleaner design. But it would have advantages even if it didn't: you have to understand a program completely to rewrite it, so there is no better way to get one loaded into your head.

Write rereadable code. All programmers know it's good to write readable code. But you yourself are the most important reader. Especially in the beginning; a prototype is a conversation with yourself. And when writing for yourself you have different priorities. If you're writing for other people, you may not want to make code too dense. Some parts of a program may be easiest to read if you spread things out, like an introductory textbook. Whereas if you're writing code to make it easy to reload into your head, it may be best to go for brevity.

Work in small groups. When you manipulate a program in your head, your vision tends to stop at the edge of the code you own. Other parts you don't understand as well, and more importantly,

can't take liberties with. So the smaller the number of programmers, the more completely a project can mutate. If there's just one programmer, as there often is at first, you can do all-encompassing redesigns.

Don't have multiple people editing the same piece of code. You never understand other people's code as well as your own. No matter how thoroughly you've read it, you've only read it, not written it. So if a piece of code is written by multiple authors, none of them understand it as well as a single author would. And of course you can't safely redesign something other people are working on. It's not just that you'd have to ask permission. You don't even let yourself think of such things. Redesigning code with several authors is like changing laws; redesigning code you alone control is like seeing the other interpretation of an ambiguous image.

If you want to put several people to work on a project, divide it into components and give each to one person.

Start small. A program gets easier to hold in your head as you become familiar with it. You can start to treat parts as black boxes once you feel confident you've fully explored them. But when you first start working on a project, you're forced to see everything. If you start with too big a problem, you may never quite be able to encompass it. So if you need to write a big, complex program, the best way to begin may not be to write a spec for it, but to write a prototype that solves a subset of the problem. Whatever the advantages of planning, they're often outweighed by the advantages of being able to keep a program in your head.

Even more striking are the number of officially sanctioned projects that manage to do all eight things wrong. In fact, if you look at the way software gets written in most organizations, it's almost as if they were deliberately trying to do things wrong. In a sense, they are. One of the defining qualities of organizations since there have been such a thing is to treat individuals as interchangeable parts. This works well for more parallelizable tasks, like fighting

wars. For most of history a well-drilled army of professional soldiers could be counted on to beat an army of individual warriors, no matter how valorous. But having ideas is not very parallelizable. And that's what programs are: ideas.

It's not merely true that organizations dislike the idea of depending on individual genius, it's a tautology. It's part of the definition of an organization not to. Of our current concept of an organization, at least.

Maybe we could define a new kind of organization that combined the efforts of individuals without requiring them to be interchangeable. Arguably a market is such a form of organization, though it may be more accurate to describe a market as a degenerate case—as what you get by default when organization isn't possible.

Probably the best we'll do is some kind of hack, like making the programming parts of an organization work differently from the rest. Perhaps the optimal solution is for big companies not even to try to develop ideas in house, but simply to buy them. But regardless of what the solution turns out to be, the first step is to realize there's a problem. There is a contradiction in the very phrase "software company." The two words are pulling in opposite directions. Any good programmer in a large organization is going to be at odds with it, because organizations are designed to prevent what programmers strive for.

Good programmers manage to get a lot done anyway. But often it requires practically an act of rebellion against the organizations that employ them. Perhaps it will help if more people understand that the way programmers behave is driven by the demands of the work they do. It's not because they're irresponsible that they work in long binges during which they blow off all other obligations, plunge straight into programming instead of writing specs first, and rewrite code that already works. It's not because they're unfriendly that they prefer to work alone, or growl at people who pop their head in the door to say hello. This apparently random collection of annoying habits has a single explanation: the power of holding a program in one's

head.

Whether or not understanding this can help large organizations, it can certainly help their competitors. The weakest point in big companies is that they don't let individual programmers do great work. So if you're a little startup, this is the place to attack them. Take on the kind of problems that have to be solved in one big brain.

Thanks to Sam Altman, David Greenspan, Aaron Iba, Jessica Livingston, Robert Morris, Peter Norvig, Lisa Randall, Emmett Shear, Sergei Tsarev, and Stephen Wolfram for reading drafts of this.

|