33

Maintain High Productivity

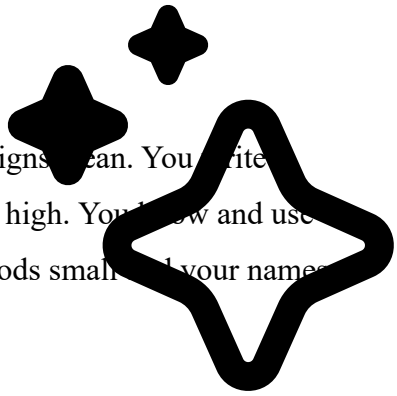
6. I will do all that I can to keep the productivity of myself and others as high as possible. I will do nothing that decreases that productivity.

Productivity. That's quite a topic, isn't it. How often do you feel that it's the only thing that matters at your job? If you think about it, productivity is what this book, and what all my books on software, is about.

They're about how to go fast.

And what we've learned over the last eight decades of software is that the way you go fast is to go well.

The *only* way to go fast is to go well.



So you keep your code clean. You keep your designs clean. You write semantically stable tests and keep your coverage high. You know and use appropriate design patterns. You keep your methods small and your names precise.

But those are all *indirect* methods of achieving productivity. In this chapter, we're going to talk about much more direct ways to keep productivity high.

Explain this concept

Highlight | Add Note | Copy

Copy

- Viscosity: Keeping your development environment efficient
- Managing distractions: Dealing with everyday business and personal life
- Time management: Effectively separating productive time from all the other junk you have to do

Viscosity

Programmers are often very myopic when it comes to productivity. They view the primary component to productivity as their ability to write code quickly.

But the writing of code is a very small part of the overall process. If you made the writing of code *infinitely* fast, it would increase overall productivity by only a small amount.

That's because there's a lot more to the software process than just writing code. There's at least

- Building
- Testing
- Debugging
- Deploying

And that doesn't count the requirements, the analysis, the design, the meetings, the research, the infrastructure, the tooling, and all the other stuff that goes into a software project.

So, although it is important to be able to write code efficiently, it's not even close to the biggest part of the problem. Let's tackle some of the other issues one at a time.

Building

If it takes you 30 minutes to build after a 5-minute edit, then you can't be very productive, can you?

There is no reason, in the third and subsequent decades of the twenty-first century, that builds should take more than a minute or two.

Before you object to this, think about it. How could you speed up the build? Are you utterly certain, in this age of cloud computing, that there is no way to dramatically speed up your build? Find whatever is causing the build to be slow, and fix it. Consider it a design challenge.

Testing

Is it your tests that are slowing the build? Same answer: Speed up your tests. Here, look at it this way. My poor little laptop has four cores running at a clock rate of 2.8GHz. That means it can execute around *10 billion instructions per second.*

Do you even have 10 billion instructions in your whole system? If not, then you should be able to test your whole system in less than a second.

Unless, of course, you are executing some of those instructions more than once.

For example, how many times do you have to test login to know that it works? Generally speaking, once should be sufficient.

So how many of your tests go through the login process? Any more than one would be a waste!

If login is required before each test, then, when executing tests, you should short-circuit the login process. Use one of the mocking patterns. Or, if you must, remove the login process from systems built for testing.

The point is, don't tolerate repetition like that in your tests. It can make them horrifically slow.

As another example, how many times do your tests walk through the navigation and menu structure of the user interface? How many tests start at the top and then walk through a long chain of links to finally get the system into the state where the test can be run?

Any more than once per navigation pathway would be a waste! So build a special testing API that allows the tests to quickly force the system into the state you need, without logging in, and without navigating.

How many times do you have to execute a query to know that it works? Once! So mock out your databases for the majority of your tests. Don't allow the same queries to be executed over and over and over again.

Peripheral devices are slow. Disks are slow. Web sockets are slow. UI screens are slow. Even SSDs can be slow. Don't let slow things slow down your tests. Mock them out. Bypass them. Get them out of the critical path of your tests.

Don't tolerate slow tests. Keep your tests running fast!

Debugging

Does it take a long time to debug things? Why? Why is debugging slow?

You are using a testing discipline and writing unit tests, aren't you? You are writing acceptance tests too, right? And you are measuring test coverage with a good coverage analysis tool, right? And you are periodically proving that your tests are semantically stable by using a mutation tester, right?

If you are doing all those things, or even just *some* of those things, your debug time can be reduced to insignificance.

Deploying

Does deployment take forever? Why? I mean, you *are* using deployment scripts, right? You aren't deploying manually, are you?

Remember, you are programmers. Deployment is a procedure; automate it! And write tests for that procedure too! You should be able to deploy your system, every time, with a single click.

Managing Distractions

One of the most pernicious destroyers of productivity is distraction from the job. There are many different kind of distractions. It is important for you to know how to recognize them and defend against them.

Meetings

Are you slowed down by meetings? I have a very simple rule for dealing with meetings. It goes like this:

When the meeting gets boring, leave.

You should be polite about it. Wait a few minutes for a lull in the conversation, and then tell the participants that you believe your input is no longer required and ask them if they would mind if you returned to the rather large amount of work you have to do.

Never be afraid to leave a meeting. If you don't figure out how to leave, then some meetings will keep you forever.

You would also be wise to decline most meeting invitations. The best way to avoid getting caught in long, boring meetings is to politely refuse the invitation in the first place.

Don't be seduced by the fear of missing out. If you are truly needed, they'll come get you.

When someone invites you to a meeting, make sure they've convinced you that you really need to go. Make sure they understand that you can only afford a few minutes and that you are likely to leave before the meeting is over.

And make sure you sit close to the door.

And if you are a group leader or a manager, remember that one of your primary duties is to defend your team's productivity by *keeping them out of meetings*.

Music

I used to code to music; long, long ago. But I found that listening to music impedes my concentration. Over time, I realized that listening to music only *feels* like it helps me concentrate, when actually it divides my attention.

One day, while looking over some year-old code, I realized that my code was suffering under the lash of the music. There, scattered through the code in a series of comments, were the lyrics to the song I had been listening to.

Since then, I've stopped listening to music while I code, and I've found I am much happier with the code I write and with the attention to detail that I can give it.

Programming is the act of arranging elements of procedure through sequence, selection, and iteration.

Music is composed of tonal and rhythmic elements arranged through sequence, selection, and iteration.

Could it be that listening to music uses the same parts of your brain that programming uses, thereby consuming part of your programming ability? That's my theory, and I'm sticking to it.

You will have to work this out for yourselves. Maybe the music really does help you. But maybe it doesn't. I'd advise you to try coding without music for a week, and see if you don't end up producing more and better code.

Mood

It's important to realize that being productive requires that you become skilled at managing your emotional state. Emotional stress can kill your ability to code. It can break your concentration and keep you in a perpetually distracted state of mind.

For example, have you ever noticed that you can't code after a huge fight with your significant other?[1] Oh, maybe you type a few random characters in your IDE, but they don't amount to much. Perhaps you pretend to be productive by hanging out in some boring meeting that you don't have to pay much attention to.

---

1. One of the wisest attitudes I've encountered regarding relationships came from the movie *The Comancheros* (1961). John Wayne's character said: "Me and my wife used to fight like a couple of wildcats with only one tree between us. But you soon find out that it doesn't make a tinker's damn who's got the upper hand. A few years roll by and you kinda settle down to being at ease with each other. Then life gets worth living."

Here's what I've found works best to restore me to productivity.

Act. Act on the root of the emotion. Don't try to code. Don't try to cover the feelings with music or meetings. It won't work. Act to resolve the

emotion.

If you find yourself at work too sad or depressed to code because of a fight with your spouse, then call them to try to resolve the issue. Even if you don't actually get the issue resolved, you'll find that the action to attempt a resolution will sometimes clear your mind well enough to code.

You don't actually have to solve the problem. All you have to do is convince yourself that you've taken enough appropriate action. I usually find that's enough to let me redirect my thoughts to the code I have to write.

The Flow

There's an altered state of mind that many programmers enjoy. It's that hyper-focused, tunnel vision state where the code seems to pour out of every orifice of your body. It can make you feel superhuman.

Despite the euphoric sensation, I've found over the years that the code I produce in that altered state tends to be pretty bad. The code is not nearly as well considered as code I write in a normal state of attention and focus.

So nowadays, I resist getting into the flow.

Pairing is a very good way to stay out of the flow. The very fact that you must communicate and collaborate with someone else seems to interfere with the flow.

Avoiding music also helps me stay out of the flow, because it allows the actual environment to keep me grounded in the real world.

If I find that I'm starting to hyper focus, I'll break away and do something else for a while.

Time Management

One of the most important ways to manage distraction is to employ a time management discipline. The one I like best is *the Pomodoro Technique*.[2]

2. [Pomodoro].

Pomodoro is Italian for tomato. Indeed, English-speaking teams tend to use the word *tomato* instead. But you'll have better luck with Google if you search for "Pomodoro Technique."

The aim of the technique is to help you manage your time and focus during a regular workday. It doesn't concern itself with anything beyond that.

At its core, the idea is quite simple. Before you begin to work, you set a timer (traditionally a kitchen timer in the shape of a tomato) for 25 minutes.

Next, you work. And you work until the timer rings.

Then, you break for five minutes, clearing your mind and body.

Then, you start again. Set the timer for 25 minutes, work until the timer rings, and then break for 5. And you do this over and over.

There's nothing magical about 25 minutes. I'd say anything between 15 and 45 minutes is reasonable. But once you choose a time, stick with that time for a while.

Of course, if I were 30 seconds away from getting a test to pass when the timer rang, I'd finish the test. On the other hand, maintaining the discipline is important. I wouldn't go more than a minute beyond.

So far, this sounds mundane, but handling interruptions, like phone calls, is where this technique shines. The rule is to *Defend the Tomato!*

Tell whoever is trying to interrupt you that you'll get back to them within 25 minutes—or whatever the length of your tomato is. Dispatch the interruption as quickly as possible, and then return to work.

Then, after your break, handle the interruption.

This means that the time between tomatoes will sometimes get pretty long, because people who interrupt you often require a lot of time.

Again, that's the beauty of this technique. Because, at the end of the day, you count the number of tomatoes you completed. And that gives you a measure of your productivity.

Once you've gotten good at breaking your day up into tomatoes like this, and defending the tomatoes from interruptions, then you can start planning your day by allocating tomatoes to it. You may even begin estimating your tasks in terms of tomatoes and planning your meetings and lunches around them.