The Pragmatic Programmer - Your Journey To Mastery, 20th Anniversary, by David Thomas and Andrew Hunt

Tip 1: Care About Your Craft
There is no point in developing software unless you care about doing it well.

Tip 2: Think! About Your Work
Think about what you're doing while you're doing it. Never run on auto-pilot. Constantly be thinking, critiquing your work in real time.

Tip 3: You Have Agency
If technology seems to be passing you by, make time to study new stuff that looks interesting. This industry gives you a remarkable set of opportunities. Be proactive, and take them.

Tip 4: Provide Options, Don't Make Lame Excuses
Instead of excuses, provide options. Don't say it can't be done; explain what can be done to salvage the situation.

Tip 5: Don't Live with Broken Windows
Don't leave "broken windows" (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered.

Tip 6: Be a Catalyst for Change
People find it easier to join an ongoing success. Show them a glimpse of the future and you'll get them to rally around.

Tip 7: Remember the Big Picture
Keep an eye on the big picture. Constantly review what's happening around you, not just what you personally are doing.

Tip 8: Make Quality a Requirements Issue
Great software today is often preferable to the fantasy of perfect software tomorrow. If you give your users something to play with early, their feedback will often lead you to a better eventual solution.

Tip 9: Invest Regularly in Your Knowledge Portfolio
Learn at least one new language every year. Read a technical book each month. Read nontechnical books, too. Stay current.

Tip 10: Critically Analyze What You Read and Hear
You need to ensure that the knowledge in your portfolio is accurate and unswayed by either vendor or media hype.

Tip 11: English is Just Another Programming Language
Know your audience. Know what you want to say. Choose your moment. Choose a style. Make it look good. Involve your audience. Be a listener. Get back to people.

Tip 12: It's Both What You Say and the Way You Say It
Unless you work in a vacuum, you need to be able to communicate. The more effective that communication, the more influential you become.

Tip 13: Build Documentation In, Don't Bolt It On
Commenting source code gives you the perfect opportunity to document those elusive bits of a project that can't be documented anywhere else: engineering trade-offs, why decisions were made, what other alternatives were discarded, and so on.

Tip 14: Good Design Is Easier to Change Than Bad Design
A thing is well designed if it adapts to the people who use it. For code, that means it must adapt by changing. Easier to Change. ETC.

Tip 15: DRY—Don't Repeat Yourself
Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. The alternative is to have the same thing expressed in two or more places. If you change one, you have to remember to change the others.

Tip 16: Make It Easy to Reuse
What you're trying to do is foster an environment where it's easier to find and reuse existing stuff than to write it yourself. If it isn't easy, people won't do it. And if you fail to reuse, you risk duplicating knowledge.

Tip 17: Eliminate Effects Between Unrelated Things
We want to design components that are self-contained: independent, and with a single, well-defined purpose.

Tip 18: There Are No Final Decisions
Instead of carving decisions in stone, think of them more as being written in the sand at the beach. A big wave can come along and wipe them out at any time.

Tip 19: Forgo Following Fads
No one knows what the future may hold, especially not us! So enable your code to rock-n-roll: to "rock on" when it can, to roll with the punches when it must.

Tip 20: Use Tracer Bullets to Find the Target
Look for the important requirements, the ones that define the system. Look for the areas where you have doubts, and where you see the biggest risks. Then prioritize your development so that these are the first areas you code.

Tip 21: Prototype to Learn
Prototyping is a learning experience. Its value lies not in the code produced, but in the lessons learned. That's really the point of prototyping.

Tip 22: Program Close to the Problem Domain
The language of the problem domain may suggest a programming solution. We always try to write code using the vocabulary of the application domain.

Tip 23: Estimate to Avoid Surprises
By learning to estimate, and by developing this skill to the point where you have an intuitive feel for the magnitudes of things, you will be able to show an apparent magical ability to determine their feasibility.

Tip 24: Iterate the Schedule with the Code
So you complete the coding and testing of the initial functionality and mark this as the end of the first iteration. Based on that experience, you can refine your initial guess on the number of iterations and what can be included in each.

Tip 25: Keep Knowledge in Plain Text
We like our plain text to be understandable to humans. You need to ensure that all parties can communicate using a common standard. Plain text is that standard.

Tip 26: Use the Power of Command Shells
Gain familiarity with the shell, and you'll find your productivity soaring.

Tip 27: Achieve Editor Fluency
The major gain is that by becoming fluent, you no longer have to think about the mechanics of editing. Your thoughts will flow, and your programming will benefit.

Tip 28: Always Use Version Control
Always. Even if you are a single-person team on a one-week project. Even if it's a "throw-away" prototype. Even if the stuff you're working on isn't source code.

Tip 29: Fix the Problem, Not the Blame
It doesn't really matter whether the bug is your fault or someone else's. It is still your problem.

Tip 30: Don't Panic
Step back a pace, and actually think about what could be causing the symptoms that you believe indicate a bug.

Tip 31: Failing Test Before Fixing Code
Sometimes by forcing yourself to isolate the circumstances that display the bug, you'll even gain an insight on how to fix it. The act of writing the test informs the solution.

Tip 32: Read the Damn Error Message
First, look at the problem. Is it a crash? What if it's not a crash? What if it's just a bad result? Get in there with a debugger and use your failing test to trigger the problem.

Tip 33: "select" Isn't Broken
Remember, if you see hoof prints, think horses—not zebras. The OS is probably not broken. And select is probably just fine.

Tip 34: Don't Assume It—Prove It
Don't gloss over a routine or piece of code involved in the bug because you "know" it works. Prove it. Prove it in this context, with this data, with these boundary conditions.

Tip 35: Learn a Text Manipulation Language
People who prefer a structured tool may prefer languages such as Python or Ruby. These languages are important enabling technologies. Using them, you can quickly hack up utilities and prototype ideas.

Tip 36: You Can't Write Perfect Software
Did that hurt? It shouldn't. Accept it as an axiom of life. Embrace it. Celebrate it. Because perfect software doesn't exist. And unless you accept this as a fact, you'll end up wasting time and energy chasing an impossible dream.

Tip 37: Design with Contracts
Be strict in what you will accept before you begin, and promise as little as possible in return. Remember, if your contract indicates that you'll accept anything and promise the world in return, then you've got a lot of code to write!

Tip 38: Crash Early
Crashing is often the best thing you can do. The alternative may be to continue, writing corrupted data to some vital database.

Tip 39: Use Assertions to Prevent the Impossible
Whenever you find yourself thinking "but of course that could never happen," add code to check it. The easiest way to do this is with assertions. In many language implementations, you'll find some form of assert that checks a Boolean condition.

Tip 40: Finish What You Start
The function or object that allocates a resource should be responsible for deallocating it.

Tip 41: Act Locally
When in doubt, it always pays to reduce scope.

Tip 42: Take Small Steps—Always
Always take small, deliberate steps, checking for feedback and adjusting before proceeding. Consider that the rate of feedback is your speed limit. You never take on a step or a task that's "too big."

Tip 43: Avoid Fortune-Telling
Much of the time, tomorrow looks a lot like today. But don't count on it.

Tip 44: Decoupled Code Is Easier to Change
If A is coupled to B and C, and B is coupled to M and N, and C to X and Y, then A is actually coupled to B, C, M, N, X, and Y.

Tip 45: Tell, Don't Ask
This principle says that you shouldn't make decisions based on the internal state of an object and then update that object. Doing so totally destroys the benefits of encapsulation and, in doing so, spreads the knowledge of the implementation throughout the code.

Tip 46: Don't Chain Method Calls
Try not to have more than one "." when you access something. And access something also covers cases where you use intermediate variables.

Tip 47: Avoid Global Data
Globally accessible data is an insidious source of coupling between application components. Each piece of global data acts as if every method in your application suddenly gained an additional parameter: after all, that global data is available inside every method.

Tip 48: If It's Important Enough to Be Global, Wrap It in an API
If your application uses a database, datastore, file system, service API, and so on, it risks falling into the globalization trap. Again, the solution is to make sure you always wrap these resources behind code that you control.

Tip 49: Programming Is About Code, But Programs Are About Data
It's almost like an industrial assembly line: feed raw data in one end and the finished product (information) comes out the other. And we like to think about all code this way.

Tip 50: Don't Hoard State; Pass It Around
If your background is object-oriented programming, then your reflexes demand that you hide data, encapsulating it inside objects. These objects then chatter back and forth, changing each other's state. This introduces a lot of coupling, and it is a big reason that OO systems can be hard to change.

Tip 51: Don't Pay Inheritance Tax
A Car may be a kind of Vehicle, but it can also be a kind of Asset, Insured Item, Loan Collateral and so on. Modeling this correctly would need multiple inheritance.

Tip 52: Prefer Interfaces to Express Polymorphism
Interfaces and protocols give us polymorphism without inheritance.

Tip 53: Delegate to Services: Has-A Trumps Is-A
Inheritance encourages developers to create classes whose objects have large numbers of methods. If a parent class has 20 methods, and the subclass wants to make use of just two of them, its objects will still have the other 18 just lying around and callable. The class has lost control of its interface.

Tip 54: Use Mixins to Share Functionality
The basic idea: we want to be able to extend classes and objects with new functionality without using inheritance. So we create a set of these functions, give that set a name, and then somehow extend a class or object with them. At that point, you've created a new class or object that combines the capabilities of the original and all its mixins.

Tip 55: Parameterize Your App Using External Configuration
When code relies on values that may change after the application has gone live, keep those values external to the app. When your application will run in different environments, and potentially for different customers, keep the environment and customer-specific values outside the app.

Tip 56: Analyze Workflow to Improve Concurrency
Concurrency is when the execution of two or more pieces of code act as if they run at the same time. Parallelism is when they do run at the same time. We'd like to find out what can happen at the same time, and what must happen in a strict order. One way to do this is to capture the workflow using a notation such as the activity diagram. We're hoping to find activities that take time, but not time in our code.

Tip 57: Shared State Is Incorrect State
The problem here is not that two processes can write to the same memory. The problem is that neither process can guarantee that its view of that memory is consistent.

Tip 58: Random Failures Are Often Concurrency Issues
Concurrency in a shared resource environment is difficult, and managing it yourself is fraught with challenges.

Tip 59: Use Actors For Concurrency Without Shared State
Actors execute concurrently, asynchronously, and share nothing. If you had enough physical processors, you could run an actor on each. If you have a single processor, then some runtime can handle the switching of context between them. Either way, the code running in the actors is the same.

Tip 60: Use Blackboards to Coordinate Workflow
A blackboard, in combination with a rules engine that encapsulates the legal requirements, is an elegant solution to the difficulties found here. Order of data arrival is irrelevant and feedback is easily handled as well.

Tip 61: Listen to Your Inner Lizard
Give yourself a little time and space to let your brain organize itself. Stop thinking about the code, and do something that is fairly mindless for a while, away from a keyboard. Take a walk, have lunch, chat with someone. Maybe sleep on it. Eventually you have one of those a ha! moments.

Tip 62: Don't Program by Coincidence
Finding an answer that happens to fit is not the same as the right answer. At all levels, people operate with many assumptions in mind—but these assumptions are rarely documented and are often in conflict between different developers. Assumptions that aren't based on well-established facts are the bane of all projects.

Tip 63: Estimate the Order of Your Algorithms
Whenever you find yourself writing a simple loop, you know that you have an O(n) algorithm. If that loop contains an inner loop, then you're looking at O $(m \times n)$. You should be asking yourself how large these values can get. If the numbers depend on external factors then you might want to stop and consider the effect that large values may have on your running time or memory consumption.

Tip 64: Test Your Estimates
If it's tricky getting accurate timings, use code profilers to count the number of times the different steps in your algorithm get executed, and plot these figures against the size of the input.

Tip 65: Refactor Early, Refactor Often
Refactoring, as with most things, is easier to do while the issues are small, as an ongoing activity while coding. You shouldn't need "a week to refactor" a piece of code—that's a full-on rewrite.

Tip 66: Testing Is Not About Finding Bugs
We believe that the major benefits of testing happen when you think about and write the tests, not when you run them.

Tip 67: A Test Is the First User of Your Code
Thinking about writing a test for our method made us look at it from the outside, as if we were a client of the code, and not its author.

Tip 68: Build End-to-End, Not Top-Down or Bottom Up
We strongly believe that the only way to build software is incrementally. Build small pieces of end-to-end functionality, learning about the problem as you go. Apply this learning as you continue to flesh out the code, involve the customer at each step, and have them guide the process.

Tip 69: Design to Test
We want to avoid creating a "time bomb"—something that sits around unnoticed and blows up at an awkward moment later in the project. By emphasizing testing against contract, we can try to avoid as many of those downstream disasters as possible.

Tip 70: Test Your Software, or Your Users Will
Make no mistake, testing is part of programming. It's not something left to other departments or staff. Testing, design, coding—it's all programming.

Tip 71: Use Property-Based Tests to Validate Your Assumptions
Once we work out our contracts (you meet the conditions when you feed it input, and it will make certain guarantees about the outputs it produces) and invariants (things that remain true about some piece of state when it's passed through a function) we can use them to automate our testing.

Tip 72: Keep It Simple and Minimize Attack Surfaces
A handful of basic principles that you should always bear in mind: 1. Minimize Attack Surface Area, 2. Principle of Least Privilege, 3. Secure Defaults, 4. Encrypt Sensitive Data, 5. Maintain Security Updates

Tip 73: Apply Security Patches Quickly
Just remember that the largest data breaches in history (so far) were caused by systems that were behind on their updates. Don't let it happen to you.

Tip 74: Name Well; Rename When Needed
When you spot a problem, fix it—right here and now. When you see a name that no longer expresses the intent, or is misleading or confusing, fix it. You've got full regression tests, so you'll spot any instances you may have missed.

Tip 75: No One Knows Exactly What They Want
Requirements rarely lie on the surface. Normally, they're buried deep beneath layers of assumptions, misconceptions, and politics. Even worse, often they don't really exist at all.

Tip 76: Programmers Help People Understand What They Want
Exact specifications of anything are rare, if not downright impossible. That's where we programmers come in. Our job is to help people understand what they want. In fact, that's probably our most valuable attribute.

Tip 77: Requirements Are Learned in a Feedback Loop
Your job is to help the client understand the consequences of their stated requirements. You do that by generating feedback, and letting them use that feedback to refine their thinking.

Tip 78: Work with a User to Think Like a User
As well as giving you insight into how the system will really be used, you'd be amazed at how the request "May I sit in for a week while you do your job?" helps build trust and establishes a basis for communication with your clients. Just remember not to get in the way!

Tip 79: Policy Is Metadata
When policy changes (and it will), only the metadata for that system will need to be updated. In fact, gathering requirements in this way naturally leads you to a system that is well factored to support metadata. Implement the general case, with the policy information as an example of the type of thing the system needs to support.

Tip 80: Use a Project Glossary
Create and maintain a project glossary—one place that defines all the specific terms and vocabulary used in a project. All participants in the project, from end users to support staff, should use the glossary to ensure consistency.

Tip 81: Don’t Think Outside the Box—Find the Box
It’s not whether you think inside the box or outside the box. The problem lies in finding the box—identifying the real constraints.

Tip 82: Don’t Go into the Code Alone
Coding in the same office or remote, alone, in pairs, or in mobs, are all effective ways of working together to solve problems. If you and your team have only ever done it one way, you might want to experiment with a different style.

Tip 83: Agile Is Not a Noun; Agile Is How You Do Things
Agile is an adjective: it’s how you do something. You can be an agile developer. You can be on a team that adopts agile practices, a team that responds to change and setbacks with agility. Agility is your style, not you.

Tip 84: Maintain Small, Stable Teams
A pragmatic team is small, under 10-12 or so members. Members come and go rarely. Everyone knows everyone well, trusts each other, and depends on each other.

Tip 85: Schedule It to Make It Happen
If your team is serious about improvement and innovation, you need to schedule it. Trying to get things done “whenever there’s a free moment” means they will never happen.

Tip 86: Organize Fully Functional Teams
Build teams so you can build code end-to-end, incrementally and iteratively.

Tip 87: Do What Works, Not What’s Fashionable
How do you know “what works”? You rely on that most fundamental of Pragmatic techniques: Try it.

Tip 88: Deliver When Users Need It
Note that being able to deliver on demand does not mean you are forced to deliver every minute of every day. You deliver when the users need it, when it makes business sense to do so.

Tip 89: Use Version Control to Drive Builds, Tests, and Releases
That is, build, test, and deployment are triggered via commits or pushes to version control, and built in a container in the cloud. Release to staging or production is specified by using a tag in your version control system.

Tip 90: Test Early, Test Often, Test Automatically
We want to start testing as soon as we have code. Those tiny minnows have a nasty habit of becoming giant, man-eating sharks pretty fast, and catching a shark is quite a bit harder. So we write unit tests. A lot of unit tests.

Tip 91: Coding Ain’t Done ’Til All the Tests Run
The automatic build runs all available tests. It’s important to aim to “test for real,” in other words, the test environment should match the production environment closely. Any gaps are where bugs breed.

Tip 92: Use Saboteurs to Test Your Testing
If you are really serious about testing, take a separate branch of the source tree, introduce bugs on purpose, and verify that the tests will catch them. When writing tests, make sure that alarms sound when they should.

Tip 93: Test State Coverage, Not Code Coverage
A great way to explore how your code handles unexpected states is to have a computer generate those states. Use property-based testing techniques to generate test data according to the contracts and invariants of the code under test.

Tip 94: Find Bugs Once
If a bug slips through the net of existing tests, you need to add a new test to trap it next time. Once a human tester finds a bug, it should be the last time a human tester finds that bug. The automated tests should be modified to check for that particular bug from then on.

Tip 95: Don’t Use Manual Procedures
Everything depends on automation. You can’t build the project on an anonymous cloud server unless the build is fully automatic. You can’t deploy automatically if there are manual steps involved.

Tip 96: Delight Users, Don’t Just Deliver Code
If you want to delight your client, forge a relationship with them where you can actively help solve their problems. Even though your title might be some variation of “Software Developer” or “Software Engineer,” in truth it should be “Problem Solver.”

Tip 97: Sign Your Work
We want to see pride of ownership. “I wrote this, and I stand behind my work.” Your signature should come to be recognized as an indicator of quality. People should see your name on a piece of code and expect it to be solid, well written, tested, and documented. A really professional job. Written by a professional. A Pragmatic Programmer.

Tip 98: First, Do No Harm
If you can’t truthfully say that you tried to list all the consequences, and made sure to protect the users from them, then you bear some responsibility when things go bad.

Tip 99: Don’t Enable Scumbags
Some inventive ideas begin to skirt the bounds of ethical behavior, and if you’re involved in that project, you are just as responsible as the sponsors. No matter how many degrees of separation you might rationalize, this rule remains true.

Tip 100: It’s Your Life. Share it. Celebrate it. Build it. AND HAVE FUN!
Envision the future we could have, and have the courage to create it. Build castles in the air every day.