Pragmatic Programmer

Andy Hunt and Dave Thomas

My top 3 messages

> When it comes to observation, fortune favours the prepared mind

Meaning: "in order to have eureka moments, your non-conscious brain needs to have plenty of raw material"

> Boiled frogs: frogs that start in cold water in a pan that starts boiling the water won't jump out in time: they don't notice the gradual increase of the temperature. Don't be a frog.

> a dead program normally does a lot less damage than a crippled one

Skills of a pragmatic programmer

> Deeply understanding that a process should be difficult or will take a long time to complete gives you the stamina to keep at it.

Learn at least 1 new language every year

Become friends with your terminal: make a project without using your mouse.

Actually practice restoring a branch, rolling back to the old version, etc: knowing it is possible and actually being able to do it are different things

State machines can be more useful than you might think, it can help drawing one out.

Tracer bullets

At darts, you use the distance to the red center to calibrate yourself. Make sure you always have something to calibrate yourself, EVERY TIME you shoot. So for example, EVERY TIME you start a project, start with the hello-world example and make sure it runs end-to-end. Now slowly add new things.
Why this is important not only for your workflow:

- Users get to see something working early, which allows them to give feedback, and keep helping you/funding you

> Developers have something to work in. The most daunting piece of paper is the one with nothing on it.

> People find it easier to join an ongoing success, show them a glimpse of the future and you'll get them to rally around

Debugging

Binary Chopping: when there is a bug somewhere in your code, chop the code or the data in halves, keep the halve that causes the bug and go on : so dont check line by line or data point for data point.

Clean code

Finish what you start; the function that allocated a recourse, should be responsible for closing it.

In your mind always think of your code as a Pipeline, maybe even try to incorporate syntax that forces you to think in a pipeline kind of way. Some languages have this 'syntactic sugar' build in already.

Inheritance Tax**:** don't use inheritance: instead use interfaces and protocols
Or use 'mixins' (inherit multiple classes, each only responsible for very small thing)

Configuration: don't share data between classes, but instead put configurations behind a simple API layer because:

- keeps your code logic seperate from config representation
- enable multiple programs to access same configuration
- should be easy to change config (maybe even with a UI)

(= configuration as a service)

But also: don't overdo it, allow yourself to hardcode things if unlikely going to change

Concurrency

Many applications need to leverage concurrency in order to be fast enough for use case.

Most important thing to make this work is the rule:

> Shared state is incorrect state

a.k.a. never share data (imagine 2 sources adding the data same time, etc)

How to do this?

- Semaphore: develop a restriction on the source, such that only 1 source can access the data at a time
- Use an agent-messaging system where there the order of events is not determined in the code, but rather a result of the messages that are sent

around.

- Use a Blackboard system: all communication is indirect via this blackboard, and information/data is only shared by putting it on this blackboard

These are all frameworks that can help to solve the shared data issue, they seem abstract but the book gives specific examples.

Programming by coincidence

The fact that your code runs/seems to be working is NOT enough because:

- It might only look like it works
- you rely on some boundary condition
- the code does not scale for multiple runs or bugs happen when multiple runs are done

If you rely on assumptions, at least document them! **Best way to do this is by adding assert statements**

Algorithmic speed

If you are not sure which big O your program is in, you can just plot the running time against the size of the input.

But don't do premature optimization, only do this when you know a (part of) the program is a bottleneck in production.

Refactoring

> Don't try to refactor and add functionality at the same time

The most important tool for refactoring is testing.
Refactor often, make it a part of your job.

Testing

Think about hardware chips: they are designed to be tested. So should your code.

Make unit-tests, property-tests (a.k.a component tests) and regression tests. They are all needed.

The job of a programmer

> That is what we [programmers] do. When given something that seems simple, we annoy people by looking for edge cases and asking about them.

> Your job is to help the client understand the consequences of their stated requirements.

It is not about thinking outside of the box: it is about finding the box. What are the real constraints?

Teams

Don't blindly follow agility frameworks: make your own. different things work for different companies.

Build teams that can build working code end-to-end. Each team needs to get feedback from the end-user.

Every code change in prod should trigger a build, and tag a new version to that.

Testing should be done automatically.