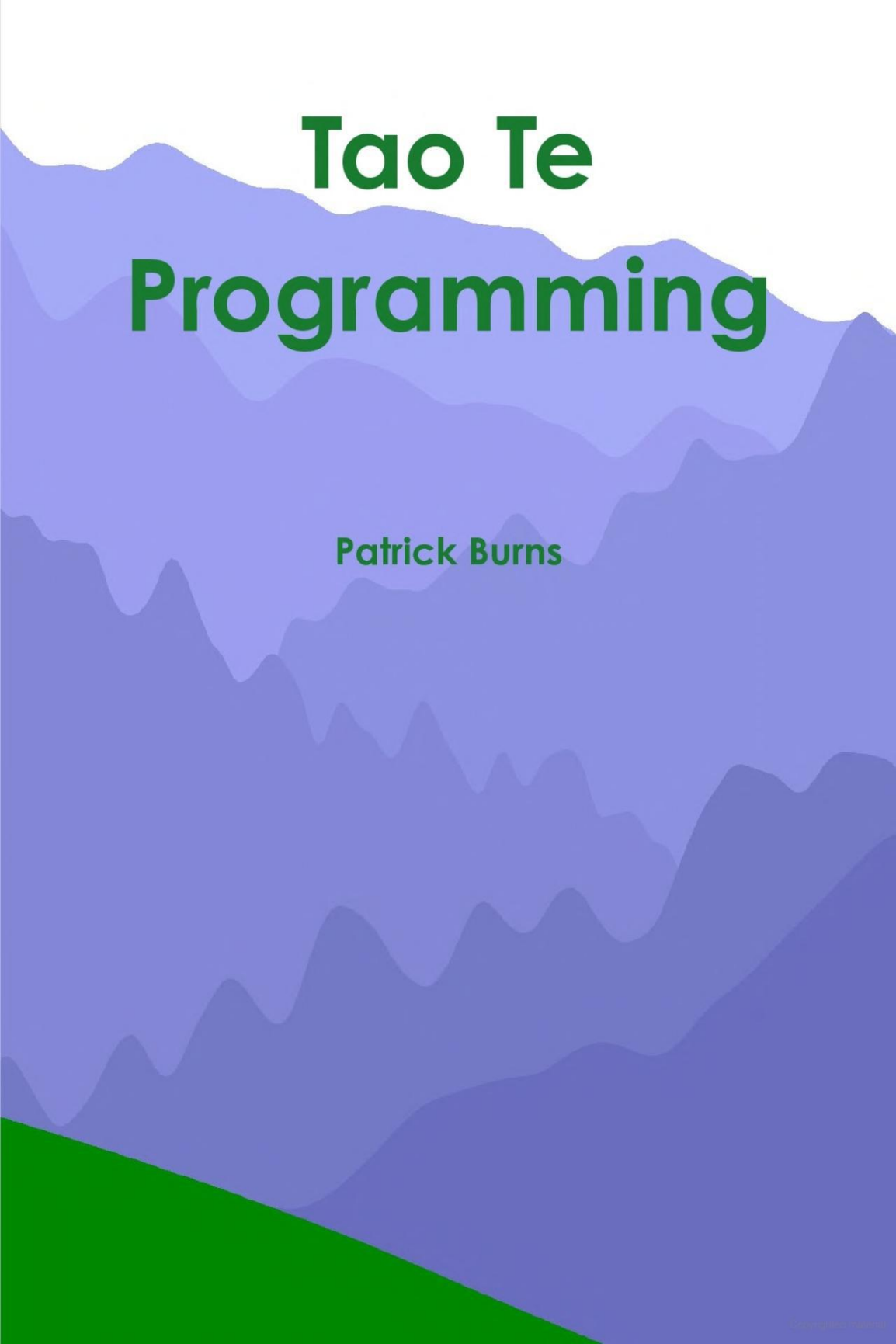
Tao Te Programming

Patrick Burns



Tao Te Programming

Patrick Burns

ISBN 978-1-291-13045-4

Contents

Reader's Guide

This is a book about what goes on in the minds of programmers. Most programming books are about the mechanics of programming. These are essential, yet they can leave novices confused and bored. *Tao Te Programming* tries to get at the spirit of programming, to expose the ways of thinking that make programming challenging and fulfilling rather than too hard and grinding.

Good programming is often about effective compromise. You can go too far in a good direction. That is why many chapters have opponents — an indication of forces you need to try to balance. Chapters can also have allies that point in a similar direction.

You can read the chapters in order. But if there is much in the book, then that something is unlikely to appear via a linear path. Bouncing around chapters is more in the book's spirit.

Artwork

The artwork was done with some simple functions in R using its random number generation.

Quotes

Unattributed quotes are from *Tao Te Ching*.

There are also snippets of creation myths scattered through the book. When you program, you create a world — just as Everything-Maker made a world. Ignore the stories if you like.

Chapter 1

Program

Computer programming is fun.
— Jon Bentley

We used to live with a cat named Esther. She taught us a proverb we hadn't known before:

Go for the hand and not the string.

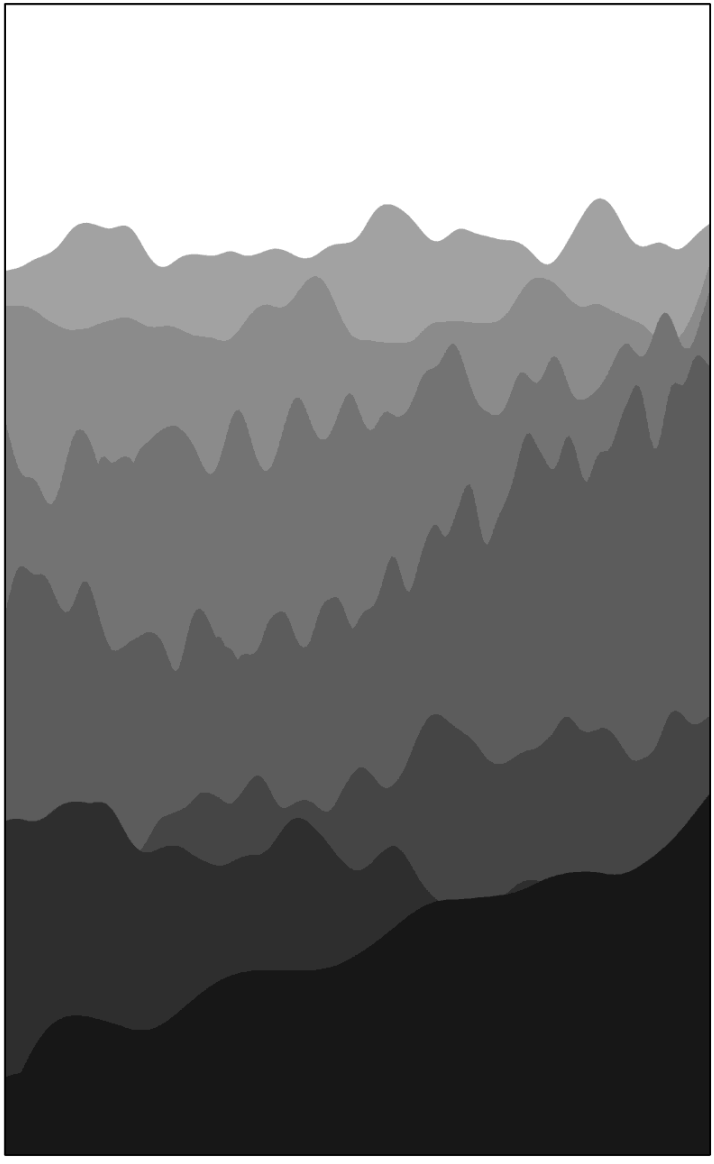
She, for one, believed in direct action.

A graphical user interface is a substitute for programming, sort of. If you are in a restaurant in a foreign country, then pointing to pictures on a card will get you something to eat. You'll survive. But learning the language will get you farther.

The main reason people don't want to program is because it is hard. Yes, it's hard. But not programming may be harder still.

The easy path looks hard.

The hard path looks easy.



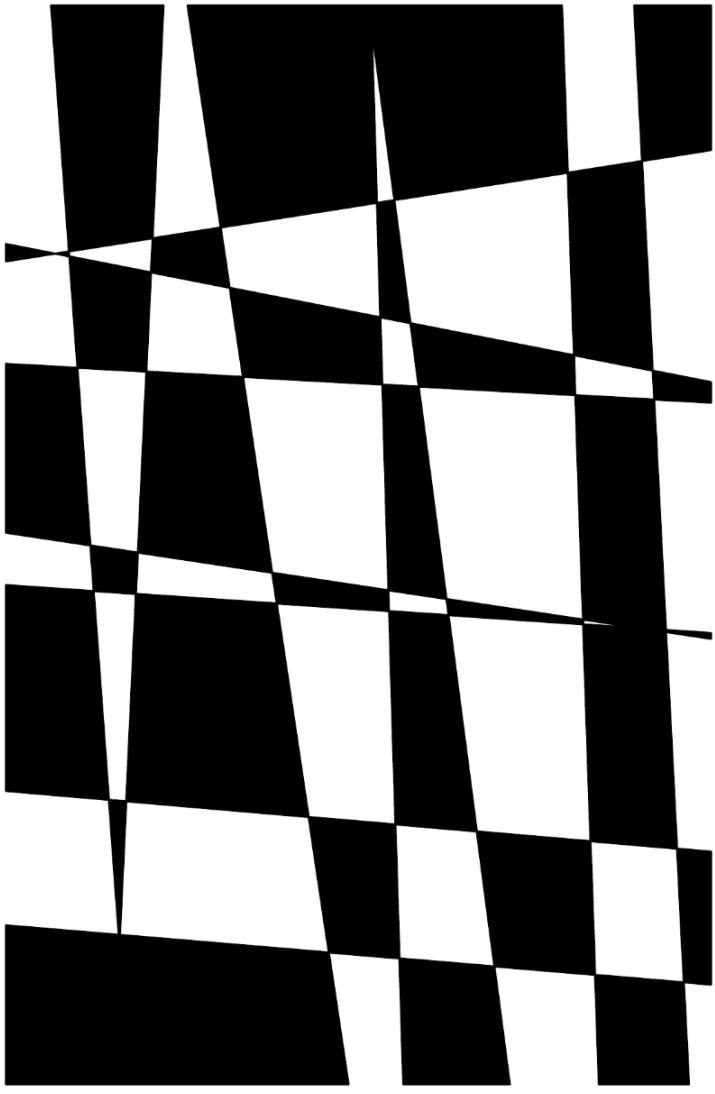
Chapter 2

Program Well

Cooks do what they do for survival. Chefs do what they do for joy. A chef works harder than a cook but is happier.

Do you want to be a cook or a chef?

Becoming a chef is a quest.



Chapter 3

Think Chess

You wouldn't think you've learned chess once you know how all the pieces move.

You don't know chess until you can think like a chess player.

This principle is even more true for programming. The important part of becoming a programmer is learning to think like a programmer. You don't need to know the details of a programming language by heart, you can just look that stuff up.

The treasure is in the structure, not the nails.

Feel Chess

Part of learning to think like a programmer is learning to feel like a programmer. Using and creating software are emotional experiences.

Much as we'd like to think that our rational thinking is the important part, the reality is that emotions dominate.

Being a good programmer is as much about emotional strength as intellectual strength.

Your first brush with programming was most likely a negative experience. Don't let that define your relationship to it.

Take Time

You didn't learn to ride a bicycle or a skateboard in one day. You'll not learn to program in one day. It takes time. Relax.

> *We may say most aptly that the Analytic Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.*
>
> — Ada Lovelace

Chapter 4

Carve Reality

There is no programming without abstraction.

Algebra is an example of abstraction. The symbol x stands not for a specific number but for some number. x is some special part of the numbers.

Words are abstractions. We carve out a piece of reality, separate it from the rest, and name it.

The name that is spoken is not the immortal name

When we speak, we string abstractions together to create a sentence or a paragraph that does what we want. Sometimes we don't have a proper abstraction — we have to make up a new word. New words are born as slang; some words live past adolescence.

Programming is the same process as speaking a natural language. The "words" are different, the syntax is different, the process is the same. If you can talk, you can program.

A difference is that slang is created much more often in programming.

Repetition is the Cue

Wherever there is repetition, there is an opportunity for abstraction.

If you repeatedly sum up numbers and divide by how many numbers there are (we call that the mean), then you should make an abstraction. Create a routine for that and call it `mean`.

A programmer's task is to:

- spot repetitions
- package each into the most appropriate abstraction

Sometimes it is easy to see the abstraction that will work best. Sometimes not.

Compression

You can think of programming as an exercise in compression.

When data are compressed, a code is created so that the actual data can be written more compactly. What's done once has to stay, but repeated parts can be shrunk. Programming is similar.

Carve and compact.

Ally

- Chapter 9: Verbalize and Nounalize
- Chapter 54: Do Not Repeat Repeat Repeat

Chapter 5

Solve the Problem

Here are four steps for general problem solving (though of course we have programming in mind).

1. List the starting ingredients
2. State the desired results
3. Break the journey from step 1 to step 2 into subproblems
4. Put the subproblem solutions together

This is a recursive algorithm — we do the same four steps on each of the subproblems, and on their subproblems.

I'm guessing that this is how almost all problems are solved — mostly subconsciously.

Breaking Up

The hard part is step 3 (but sometimes step 2 is murky). This is another case of abstraction, of carving up reality. There may be multiple possible combinations of subproblems — your task is to create a reasonable combination.

Avoid feeling that you must solve the whole thing in one go. That's the recipe for being overwhelmed and stuck.

Our natural inclination is to go from beginning to end when breaking up a problem. If that is not bearing fruit, try working backwards from the end.

The more you practice breaking a problem into subproblems, the better you get at it. This is important enough to practice deliberately.

This section should be surrounded by flashing lights. Breaking a problem into pieces seems to be the central block with programming for most people.

Great acts are done by a series of small deeds

Surprise is Good

The act of breaking the problem apart can highlight connections.

If It's Not Working

Two things to try if you are not getting your problem solved:

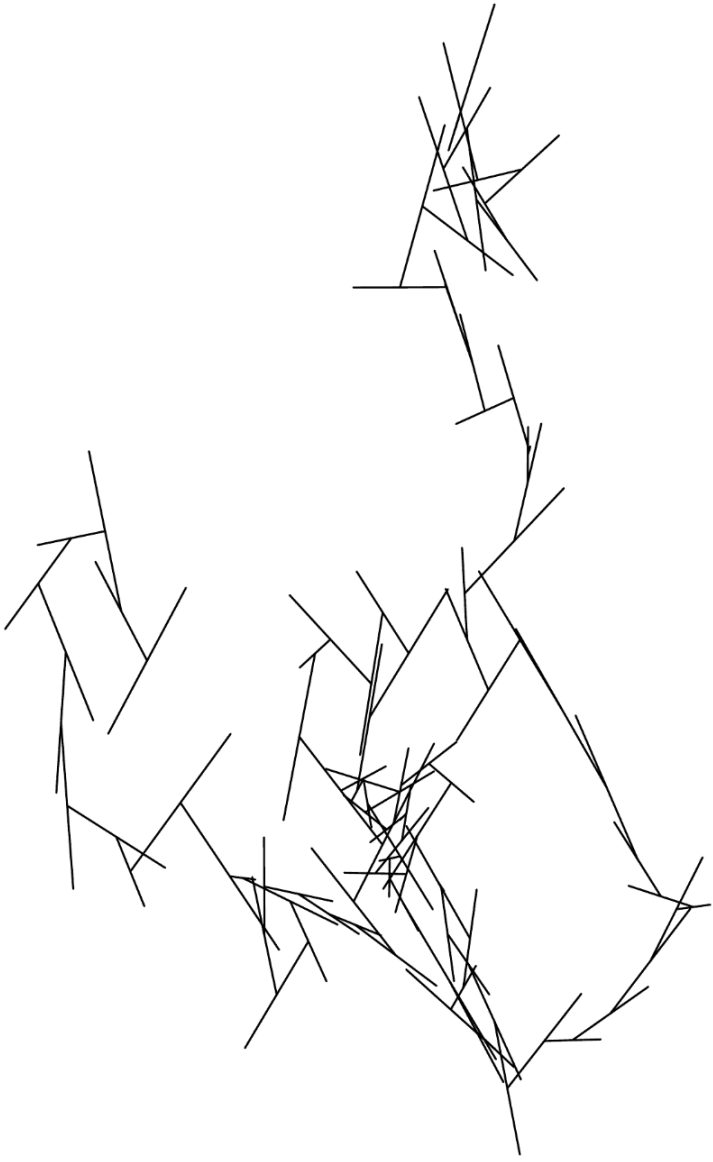
- walk
- sleep

Humans evolved as nomads, our brains work best when we are walking.

Sleep allows us to lay down memories, sort possibilities, and have a brain ready for active duty.

Opponent

- Chapter 6: Don't Solve the Problem
- Chapter 55: Climb Above the Solution
- Chapter 41: Give Up Control



Chapter 6

Don't Solve the Problem

We always want to jump in and attack problems directly.

However, the real problem is often not what we first think. This is especially true when we're solving someone else's problem. It is ever so common for someone to ask for some specific technical thing which turns out not to address their actual concern, or to be a poor approach to it.

Before you dig into a problem, make sure you know what the problem is. I count three possible outcomes:

- the real problem is easier than the original — you have saved time and trauma
- the real problem is harder than the original — but you've avoided working on the wrong one

- the original problem is the real problem — but now you can tackle it with confidence

Refuse the Part

If you are cast as the programmer who will solve the problem, you don't have to accept the role. Sometimes the best solution is no programming at all. Cast yourself as the non-programmer who solved the problem.

"Can you create a program for me to improve this calculation?"

"I could, but I won't. You need to do something else entirely."

The sage stays behind in order to be ahead

Surprise is Bad

Learning that you've just spent a lot of time solving the wrong thing is disheartening.

Opponent

- Chapter 5: Solve the Problem

Ally

- Chapter 55: Climb Above the Solution

Chapter 7

Enjoy Confusion

If you are confused, then you know something now that you didn't before.

Our instinct is to think of confusion as a negative state. We want to jump back to certainty as quickly as possible — and we don't care which way we jump.

Better to savor confusion. Only in confusion can our wrong assumptions dissolve away.

Darkness within darkness, the gateway

Surprise is Good

If you're surprised, a door has been opened to a larger world.

Speaking in Signs

The Navajo tell of the start of this world — there were three previous worlds. People were driven from each of the other worlds because they quarrelled so much.

Four mysterious beings approached the people. They tried to instruct the people through signs but without speaking. The gods tried for four days without success. On the last day Black Body, the god of fire, stayed behind and spoke to the people in their own language.

Ally

- Chapter 17: Use Your Frustration
- Chapter 10: Pay Attention to Attention

Chapter 8

Procrastinate

Procrastination gets very bad press. Pretty much everyone seriously bullies it.

Not me. Put off a step you find hard to do. Work on the easy things first.

You will be working on the hard step subconsciously. When you return to it, then probably one of two things will happen:

- You now see how to do it
- You now see that it is the wrong thing to do — that the motto "if it's hard, it's wrong" applies

You may not have got that far if you had stayed and beat your head against the hard step. Plus the easy stuff is done.

Can you wait until the mud in the stream settles?

There are times when procrastination is absolutely, positively the wrong thing to do — see the Show Stoppers chapter.

Staircase Syndrome

When I worked in an office, I had a recurring experience. As I walked down the stairs leaving work, I would suddenly realize how to solve some problem that I'd been working on that day. The consistency of the phenomenon meant it was not coincidental.

My unconscious mind had been working on the problem. It had solved the problem. But there had to be space in my conscious mind to let the solution come to the fore.

The two steps to get your unconscious mind to do your work:

- Start the unconscious mind by consciously working on the problem
- Relax so the solution can float to consciousness

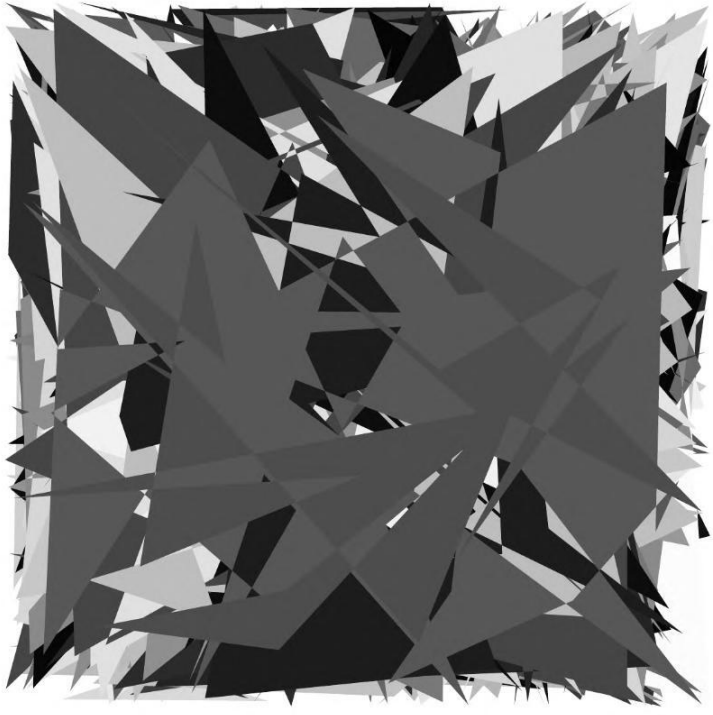
Sleep can be a good time for this: think hard about a problem just before you go to sleep; your mind is naturally empty when you wake up.

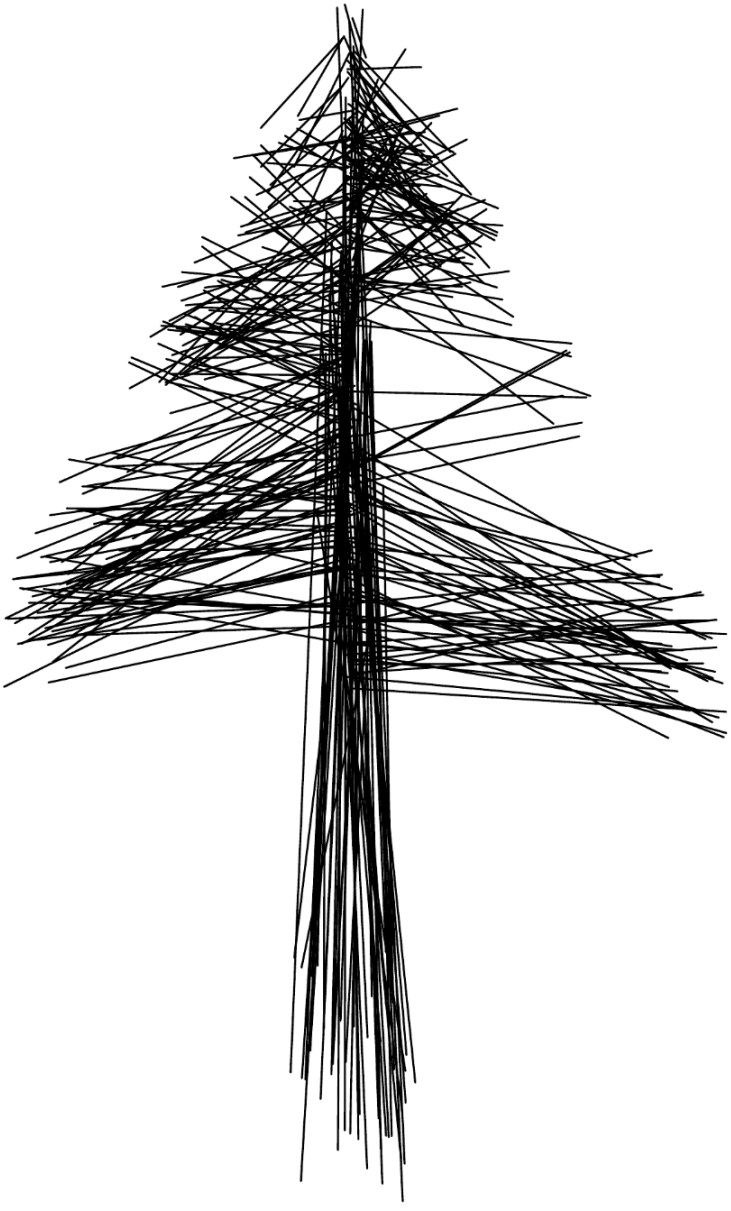
Opponent

- Chapter 66: Sidestep Show Stoppers

Ally

- Chapter 26: Be Lazy





Chapter 9

Verbalize and Nounalize

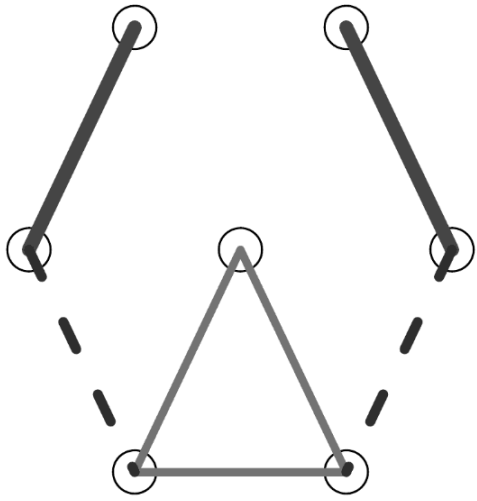
Use the concepts of nouns and verbs to create abstractions.

Nouns are stuff, data. Many times it is fairly unambiguous how to split the stuff into separate objects. (But little tweaks to data structures can be massively simplifying.)

The system is going to perform actions on these objects — these actions are the verbs. Create a set of verbs and nouns to make the whole system as simple as possible. Perhaps you might think of the system as a network where the nouns are the points and the verbs are connections between the nouns.

Make that picture simple, intuitive, pretty.

Good nouns make good verbs.



Subset

Suppose you have a subset from some given universe. Two ways of representing that subset are:

- (variable length) list of items in the subset
- (fixed length) vector indicating which items in the universe are in the subset

These both contain the same information. They will not be equally useful for a particular situation. One may be substantially better than the other. Which is better depends on how the information needs to be used.

If the items need to be in order, then the fixed length approach will be better if the subset is not too small relative to the size of the universe. If the subset is small relative to the universe, then sorting will be easier than running through the whole universe.

If the order of the items matters, then the fixed length object can be modified to contain that information, but the list of items is most likely to be the better choice.

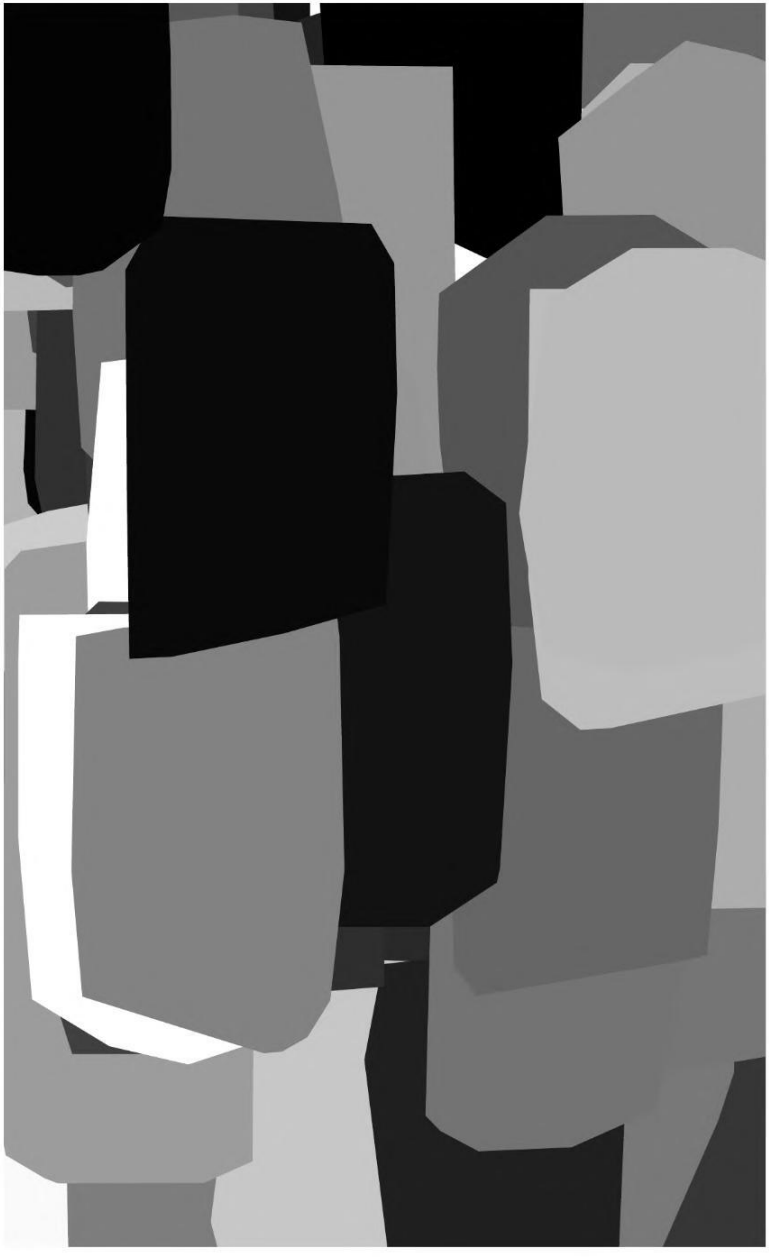
Choose the noun depending on the verb.

Cloud

Benjamin Whorf reported that in Hopi things that have duration shorter than a cloud, like lightning and meteor are verbs. Whatever lasts longer is a noun.

Ally

- Chapter 4: Carve Reality
- Chapter 30: Relish Magic



Chapter 10

Pay Attention to Attention

People have four basic levels of attention:

- adrenaline-fueled fight or flight (panic)
- alert
- half asleep
- completely asleep

Your Attention

When you first start learning a new programming language, you may easily flip between panic and inattention. Staying properly alert might be like walking a razor's edge. With persistence you can transform that razor's edge into a wide path.

Your Users' Attention

You want to do all you can to keep your users out of panic mode.

Write your code assuming that your users are inattentive. This means all the consistency you can muster, and unambiguous names.

Delicately, like frying a small fish

Transformer

I was walking with someone along a neighborhood street. A bunch of crows started raising quite a noise — clearly something was up in the crow world. Then there was a very loud pop behind us. We turned and saw a crow falling from a smoking transformer.

Did it panic and land on the transformer? Did it not know the danger? Was it a dare? Was it suicide? What were the others saying?

Ally

- Chapter 7: Enjoy Confusion
- Chapter 17: Use Your Frustration
- Chapter 29: Be Consistent
- Chapter 31: Tell a Good Story

Chapter 11

Be Accident Prone

The fraction of good things I've done that didn't arrive by accident is frighteningly small.

If you make a mistake, wonder how you can benefit from it:

- How can this be made into an even better mistake?
- How could this have been right?
- How can such mistakes be blocked?

Heuristic Algorithms

Evolutionary algorithms and relatives use randomness to do optimization. They typically use no information about the problem being solved other than the result given the inputs. Randomness provides, eventually, better solutions. On some problems they work remarkably well.

You can be your own random heuristic optimizer. You just need the courage to use your mistakes.

Bose-Einstein Mistake

Apparently the collaboration between Bose and Einstein was precipitated by a trivial error (by Bose) that allowed him to see a deeper truth.

Opponent

- Chapter 48: Be Wary

Ally

- Chapter 49: Lose Every Battle

Chapter 12

Conquer Time

You can let time just flow past you, or you can collect it in buckets.

Version control keeps track of the changes to files. Sounds simple — it is. But it is very useful.

- you can confidently experiment with code, knowing that you can put it back to its current state
- you can return to a previous version if the new version has a problem
- you can see if a particular command would have been correct in a previous version
- ...

Version control is useful for documentation and test suites as well as code. In fact it can be useful for projects that have nothing at all to do with programming.

Version control gives you a history of when and why things were changed. Of course you have to state why when you "check in" a change. Make the statement explicit, you'll thank yourself later.

You don't want to throw different types of changes into one revision. Better is to do a revision for each type of change. If you are fixing two unrelated bugs and improving the layout, then do three different revisions.

Surprise is Bad

The surprise that your new version doesn't work is extremely unpleasant if you can't rewind to the version that does work.

Chapter 13

Learn the Local Jargon

When you come to a new language, it is beneficial to learn the jargon specific to that language. Many words have very specific meanings.

For example C and C++ have "arrays", so does R. But the meaning of "array" in R is different than that in C and C++. R has the same concept as "array" in C, but uses a different word.

If you know a language's jargon, then:

- you can communicate with the other people using the language
- you will be less confused about the language
- you will better pick up the nuances of the language

A little time invested studying the vocabulary pays big dividends.

Alpha versus Beta

You almost surely know that a beta version of software is a test version that may or may not be ready for prime time. It has come to my attention that the meaning of "alpha version" is not so well known. Is an alpha version better or worse than a beta version?

It could be better: the alpha dog is better than the beta dog.

It could be worse: alpha comes before beta.

The programming definition is that an alpha version is the first thing out of the box. Often an alpha version is little more than a proof of concept. I've seen alpha versions that weren't even that. (These have ranged from no reasonable concept to merely no proof.)

Once the code is close to release-quality, it is designated beta and opened up to testing by a wider audience. What would logically be called the gamma version is just called the release.

Surprise is Bad

You don't want to be surprised by the meaning of words.

Ally

- Chapter 78: Be a Polyglot

Chapter 14

Accept Numerical Reality

Every day around the world people find things like:

```
0.1 + 0.1 + 0.1 == 0.3
```

to be false and think they are seeing a bug. They are not.

While the above equation is logically true, it depends on the numbers being exact. Computers can deal with (smallish) integers exactly, but they can not represent arbitrary numbers exactly.

Percentages

Here's what's going on. Suppose we have the counts: 23, 47, 13 and we want to turn those into percentages of total count. If we round the percentages to one decimal place, we have:

```
27.7 56.6 15.7
```

These sum to 100%.

Now suppose we round to even percent:

```
28 57 16
```

These sum to 101%.

There is no error in our calculation in the sense of a bug. There is error in our calculation in the sense of *numerical error*.

Our restriction to use only integer percentages limits our ability to produce exact results. Programs use floating point numbers which have a similar sort of restriction.

> *10.0 times 0.1 is hardly ever 1.0.*
> — Kernighan and Plauger

Here be bugs. Naive users think there are bugs when conceptually equal values are not equal. The real bugs are actually when programs assume conceptually equal values are equal.

Do not expect exact equality with floating point calculations. Use a suitable tolerance instead.

To clarify: it is possible to do calculations that are arbitrarily close to exact, but they are only practical in specific circumstances.

Vocabulary

If you are dealing with numerical algorithms, then you may need to learn some words, such

as “overflow”, “underflow” and my personal favorite “negative zero”. (Since floating point numbers are really ranges rather than points, some pedants think a minus sign on zero can mean something.)

Reality

Novices assume their ideal of numbers is implemented in the computer. The numbers they get are usually close enough to maintain that illusion.

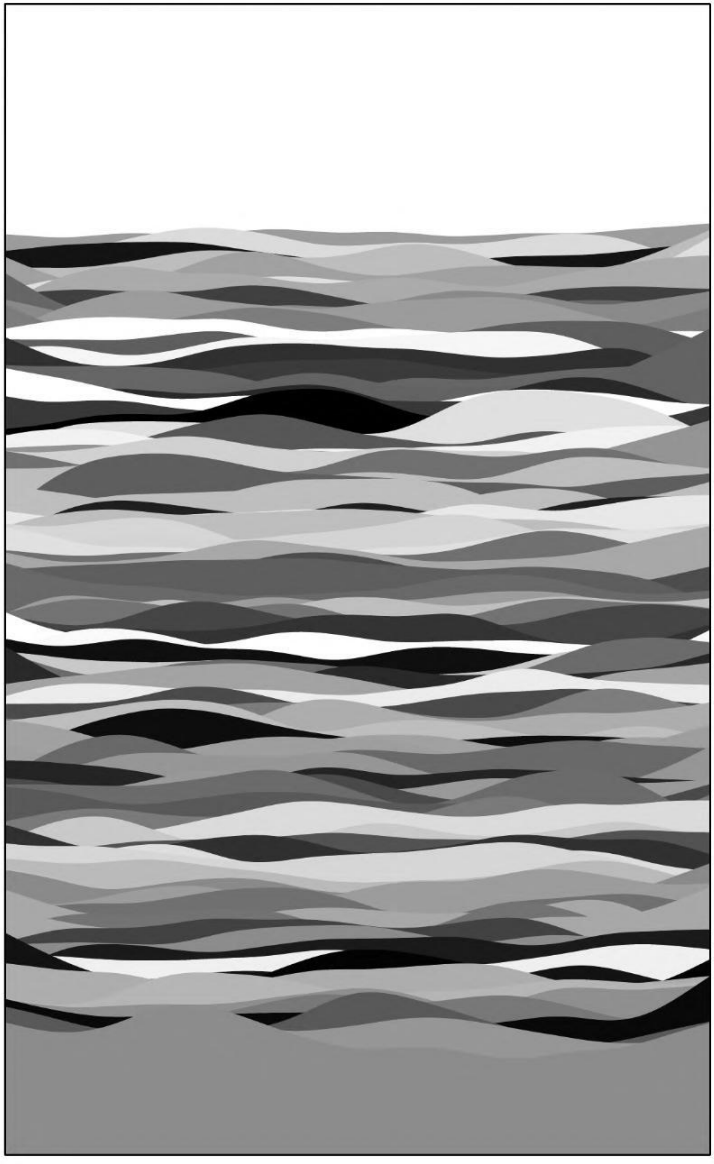
Such problems are not restricted to novices and numbers. There can be a gap between the ideal of a concept and the implementation of that concept.

If reality is:

```
implementation != concept
```

then avoid the mental image:

```
implementation == concept
```



Chapter 15

Be Stateless

A piece of code can do one or both of:

- return a value (or values)
- create one or more side effects

If you have a `mean` function, it most likely returns a number (which is the mean value of the numbers in the input).

A `print` function likely has the side effect of printing something somewhere. A `plot` function most likely has the side effect of creating a graphic of something somewhere. Another common side effect is for some object to be changed.

Return values are passed from one piece of code to another piece of code. A side effect is something actually changing.

If there were no side effects, then we humans would not be interested. Values might be passed around the code, but we'd never see anything

happen. However, we want to limit how and when side effects occur. If side effects happen willy-nilly, the system will be too complex to understand.

Most likely our `mean` function returns one numeric value with no side effects if the input is suitable. If the input is not suitable, it will probably have the side effect of throwing an error; or perhaps it will throw a warning (a side effect) and return something, a missing value possibly.

It is possible (in many languages) for `mean` to return a value but also to have the side effect of changing the input data. That is highly unlikely to be good. Best to avoid modifications if possible.

Why Reboot?

There's the story of the programmers whose car quit. Their way to try to get it going again was to all get out of the car and then get back in again.

Why does that work with computers?

Reboots fix problems of state. The state of a computer system is the result of side effects.

My computer has the state that it knows what is controlling the location of the cursor. I know that as my mouse. Another state is the shape of the cursor, which is determined by a property of whatever the cursor is over.

If a state somehow gets corrupted, then states that depend on that state will be confused and possibly become corrupted as well. States seldom heal themselves.

State

If possible, avoid state.

If not, make the state as simple as possible.

Global variables are a form of state — one that can often be avoided.


image
not
available