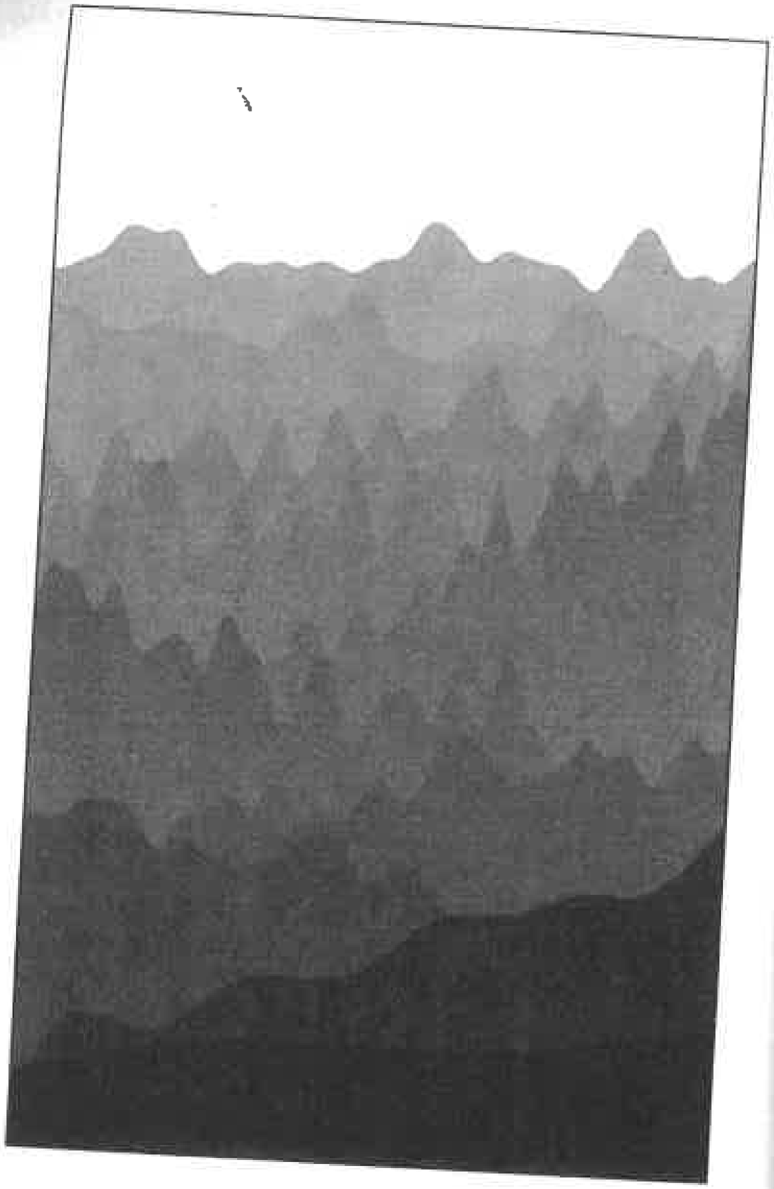


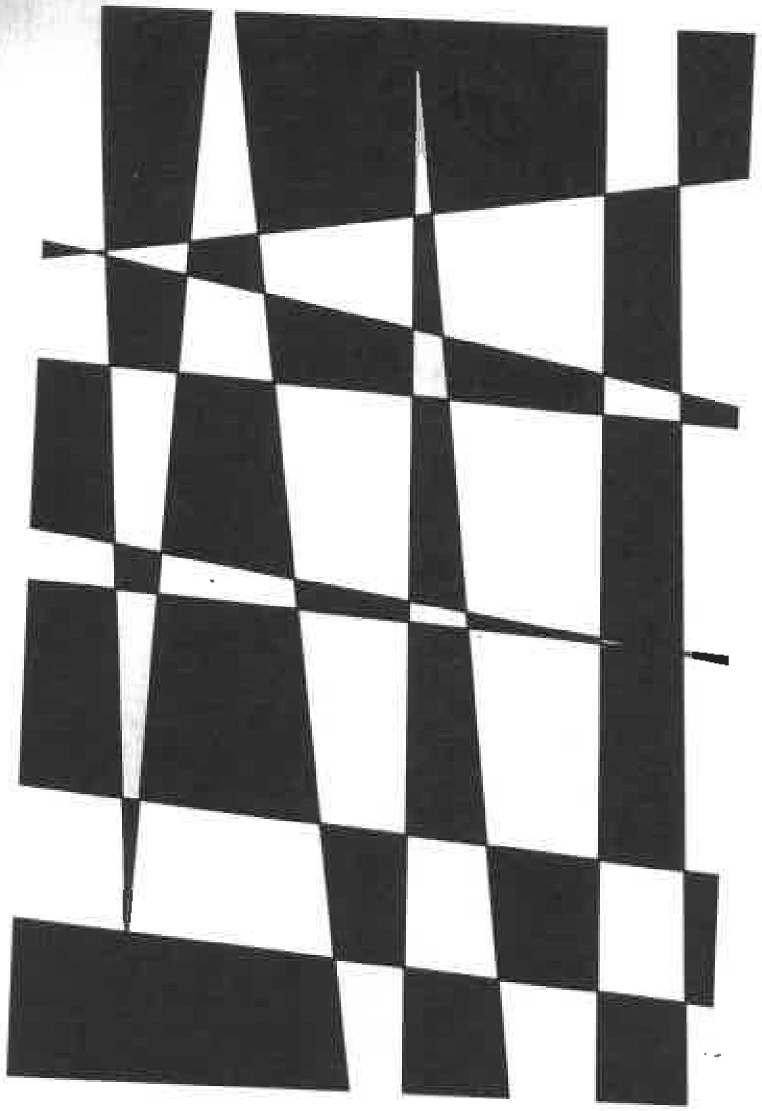
Chapter 2

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.

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

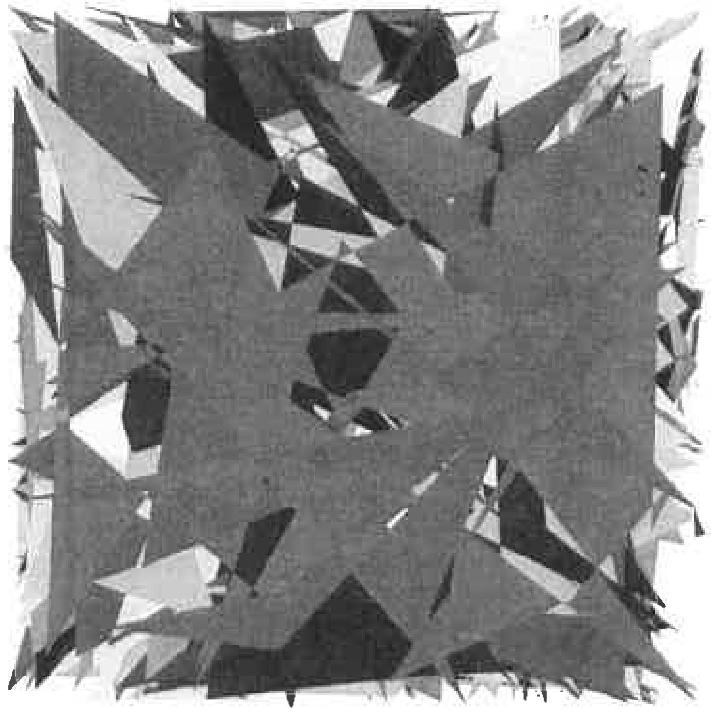
Sleep can be a good time for this: think hard about a problem just before you go to sleep; your mind is naturally empty when you wake up.

Opponent

- Chapter 66: Sidestep Show Stoppers

Ally

- Chapter 26: Be Lazy



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.

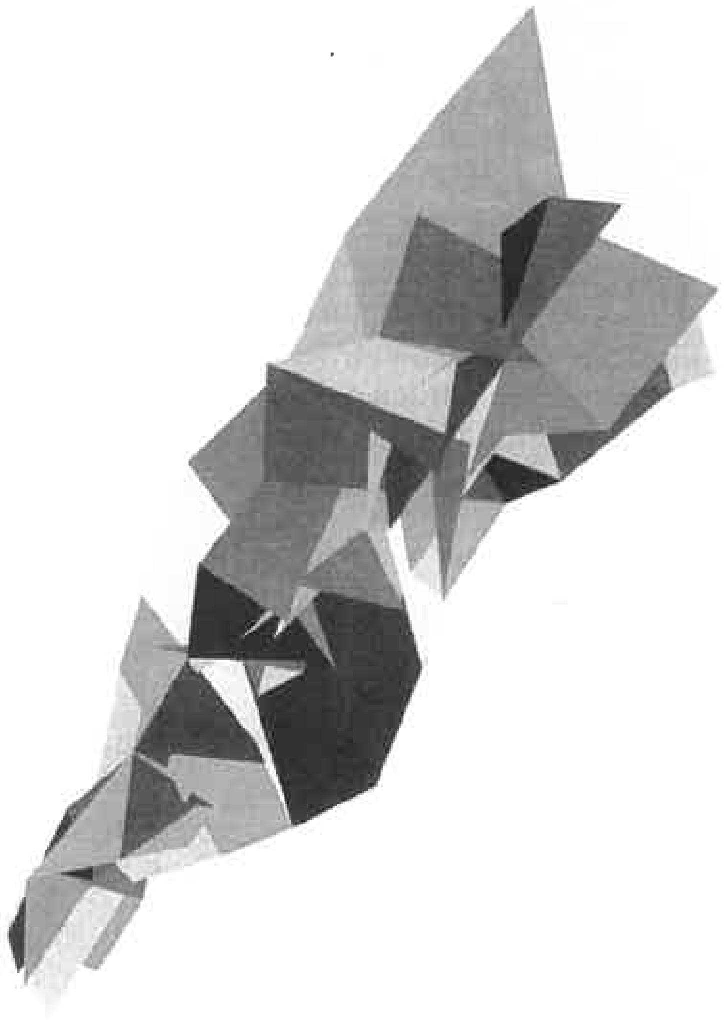
Such problems are not restricted to novices and numbers. There can be a gap between the ideal of a concept and the implementation of that concept.

If reality is:

```
implementation != concept
```

then avoid the mental image:

```
implementation == concept
```



Chapter 16

Travel in Space

As a novice programmer you can safely ignore what goes on under the hood. But to mature, you need to pay attention.

Computing is very much about memory, and memory has addresses.

Think of memory as a very long street, and each plot of land adjacent to the street has an address. The two main streets are RAM and disk — RAM generally being what needs thought.

Some of the commands in your program will demand memory. It will only get continuous blocks of addresses. (So it's easy — you only need the starting address and the ending address.) Your program may also let go of memory that it had previously grabbed.

In the beginning the street is empty. After some use there are occupied blocks interspersed with unoccupied blocks.

If your program asks for a big block of memory and there is no unoccupied block that large, then you have a problem. The operating system may figure something out for you, or it may tell you you can't do that. (Some of you may know the phrase "Go fish".)

One way of increasing the possibility of hearing "go fish" — that's a bad thing — is to fragment memory. That is, to leave occupied blocks of memory scattered throughout so there are no large unoccupied blocks.

A really good way to fragment memory — a bad thing — is to grow the size of your demands for memory. If you are in a loop and on each iteration you demand a bigger block of memory for some item, then it will have to move the memory for that item to a new, bigger space. This is not only a memory burden, but a time burden as well since it takes time to do the memory copy.

Paging

If you fill up RAM, then you will experience paging (unknowingly?). This archives items on disk so that they can be temporarily moved from RAM.

The phenomenon is of time as well as space. Virtually all of the effort can go into trying to solve the memory shuffling problem. Meanwhile only a slight amount of actual work gets done. The time to solve a problem can jump abruptly

as you increase its size.

If you experience paging, there may be a few things you can do:

- get a bigger machine (not always as stupid of a suggestion as it seems)
- break your problem into pieces (or make it smaller altogether)
- make your code more memory efficient

Memory Dance

When the code you write is executed, it performs a memory dance. It will grab memory, it will give some back, it will grab some more.

Learn to see the dance. Make the dance pretty.

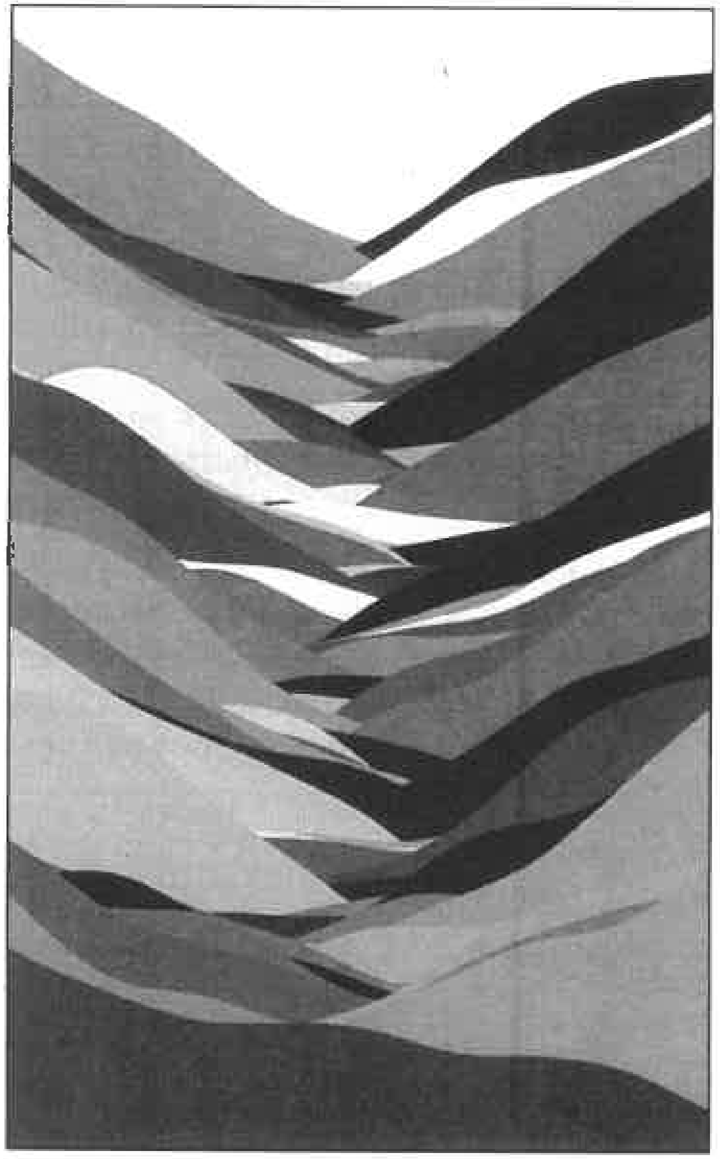
Chapter 18

Be a Hacker

The word "hacking" is predominately used to refer to the act of breaking into computers, generally for nefarious reasons. But there is another meaning of the word in which the activity is socially acceptable and very useful. The clinical synonym is "experimentation".

When experienced programmers face something they are not familiar with, their initial reaction tends to be to hack it. That is, they play around with it until they figure out how it works.

Rather than being intimidated by their ignorance, they treat the situation as a puzzle.



Chapter 25

Be Poetic

Do a lot with a little.

Seek the fruit and not the flower

Poetry and Programming

Poetry and programming are often compared. I've used the analogy myself.

There are similarities:

- measured by the line
- simple rules
- complex results
- subtleties matter

But there are also differences:

- poetry is single purpose, code is dual purpose
- repetition in poetry can be good, repetition in code is bad
- ambiguity in poetry is good, ambiguity in code is bad
- surprise in poetry is good, surprise in code is bad

Purpose

The primary purpose of poetry is to create emotional responses.

Poetry is aimed at the primitive portions of the brain that evoke emotion. Bad poets think they can directly achieve emotional response. Actually it is necessary to enlist the cognitive brain and make a flanking move on the emotions.

The purposes of computer code are:

- Create a response in a machine
- Communicate to people

If programming were only about the first, then all programs would be in assembly language.

Novices should mostly worry about the machine response. Beyond novice the emphasis is on communication. When you code, one of the

people you are most likely to want to communicate with is yourself at a future date.

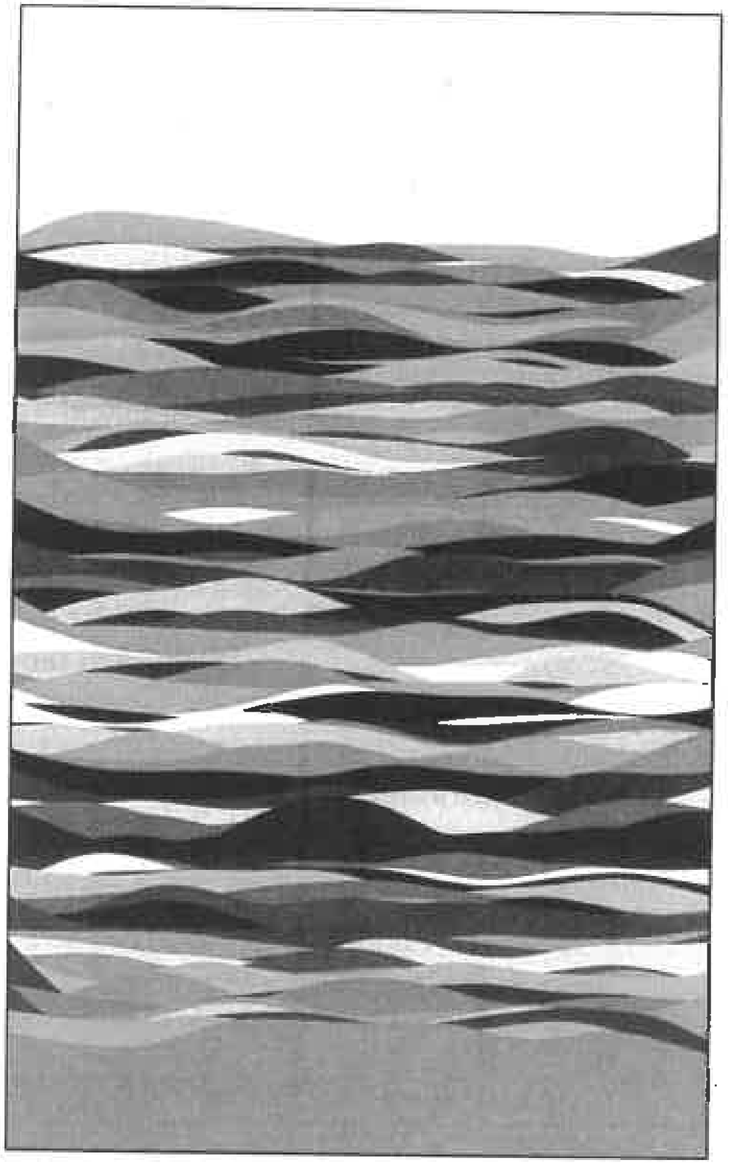
Serving two masters is hard. That is one reason that programming is particularly challenging. Balance is needed.

Opponent

- Chapter 79: Avoid Perfect

Ally

- Chapter 21: Decontaminate
- Chapter 22: Be Miserly



Chapter 26

Be Lazy

Larry Wall — who created Perl — told us of the three great virtues of programmers:

- Laziness
- Impatience
- Hubris

Laziness is virtuous in several ways. Larry emphasized exerting great effort in order to minimize overall energy expenditure — diligently writing code and documentation now in order to reduce work later.

My favorite form of laziness is to refuse to type a line of code until the time is ripe.

I had an addition to make to my code. It was a seemingly trivial addition. If the functionality were not so general, then it would have been trivial. But complexity got in the way. I spent a week not making the addition. I was the boss, I could do that.

It would be a rare company where I as a programmer would have the freedom to do "nothing" for a week.

Some people have all the hurry in their feet.

It is less total work to create a great solution than a mediocre solution.

Coyote Plants Corn

This is a story from the White Mountain Apache.

Coyote got some corn seeds. Growing corn is a lot of work. Other people plant their corn, they grow a lot more corn, and then they have to cook it all. But Coyote had a better idea. "I'll be smarter than they are", says Coyote, "I'll cook it first, and then I won't have to cook it afterwards."

Coyote was very annoyed when his corn did not grow.

Ally

- Chapter 8: Procrastinate

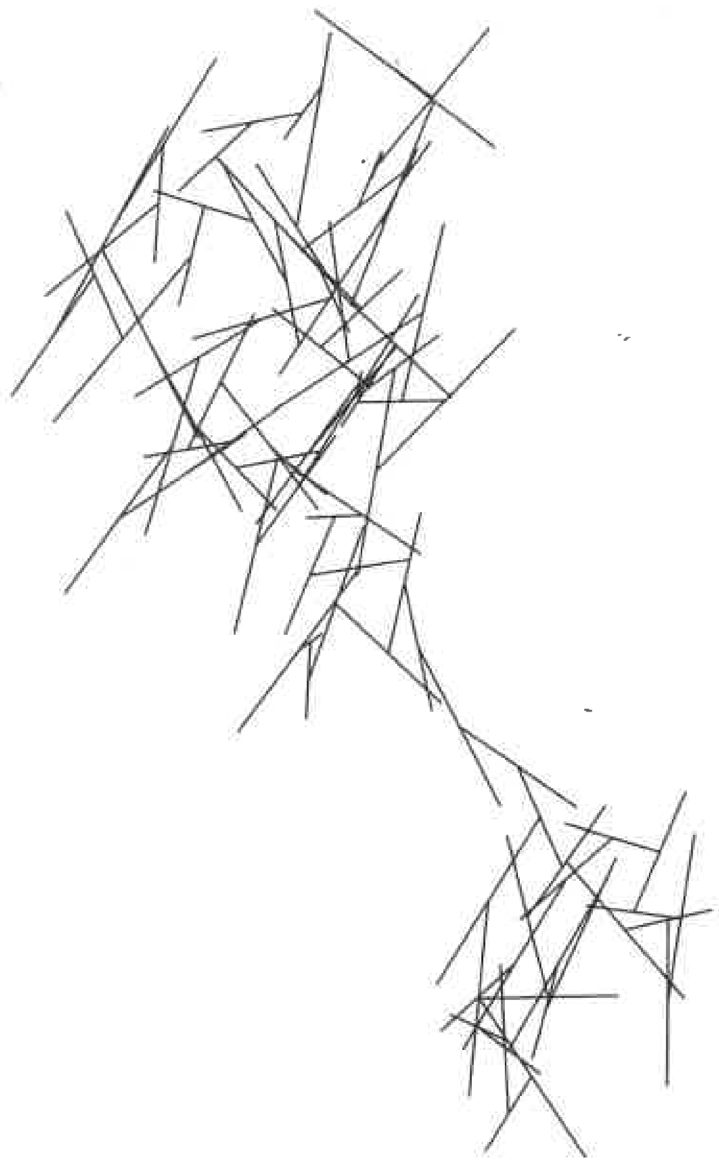
Chapter 27

Be Impatient

Impatience is the ability to envision the grand things that a computer can do, to be anxious to have the computer do all of our work.

If it's boring, then you should be wondering how to get a computer to do it.

It is ultimately faster to create a great solution than a mediocre solution.



Chapter 28

Have Hubris

Hubris is having the chutzpah to think that you can really pull off coding the grand things that a computer can do.

It is more daring to create a great solution than a mediocre solution.

Coyote Climbs to the Sky

This is from a story of the Wasco.

The 5 wolf brothers wanted to get to the sky. So Coyote shot an arrow up that stuck in the sky. Then he shot another arrow that stuck in the end of the first arrow. After a long time Coyote had a chain of arrows that stretched down to the ground. Then the wolf brothers, the dog of the eldest brother, and Coyote all climbed to the sky.

Opponent

- Chapter 79: Avoid Perfect
- Chapter 53: Beware Easy

Chapter 29

Be Consistent

The best thing you can do for your users (and yourself) is to be consistent in all aspects of your programs.

- be consistent with argument names (if users already have expectations, pay attention to those)
- argument order should be consistent
- output should be consistent — both values and side effects
- function names should be consistent

When users are half asleep, you still want them to do the right thing.

Before you let go of any software, do **everything** you can to make it as consistent as you can with the environment it is in. A little thought at the start is priceless.

This is **really** important. Seriously.

Surprise is Bad

Really bad.

Object Names

Even consistency in trivial things can be highly useful.

I often interactively create objects that are the results of various combinations of inputs. My naming convention in some instance might be something like:

```
result.a.10
result.a.05
result.b.10
```

Because of the convention it is easy to do operations on collections of those objects. You just need to use regular expressions like:

```
"result.a.*"
"result.*.10"
```

as the starting point.

Obey Expectations

Obeying user expectations is a specific form of being consistent.

Surprising your users is not going to make them happy.

Exception

Violating expectations can sometimes be good. The gain has to be much greater than the misery you impose on the users.

`data.table` (an alternative data structure) is mostly compatible with the rest of R, but it forces users to do a few things differently. The average user is not keen on being forced to learn new stuff. But tell them that their computation with a large dataset can go from taking hours to taking seconds, and they are probably willing to exert a little effort.

Pedestrian Crossing

In the UK there are 4 states of the signs for pedestrians: steady green, flashing green, flashing orange, steady orange. I would have thought that the meanings of these would be:

- steady green: safe
- flashing green: safe, but not for long
- flashing orange: caution
- steady orange: danger

The actual meanings are:

- steady green: safe

- flashing green: danger
- flashing orange: caution
- steady orange: danger

Kittens

A group of Russian school children were visiting England. They couldn't read English so in the grocery store they deciphered what was in packages by what was pictured on the outside, like canned peaches and canned pineapple. They became very upset when they went down an aisle containing packages with pictures of kittens and puppies.

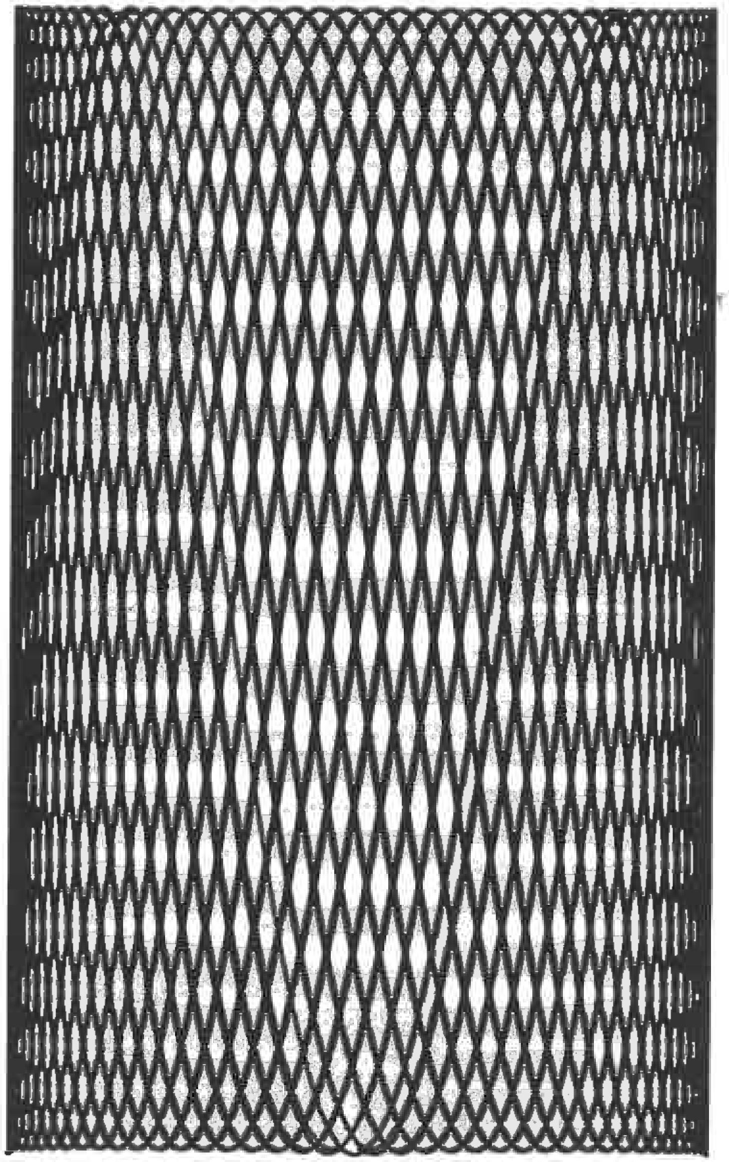
Don't depend on your users knowing the native language.

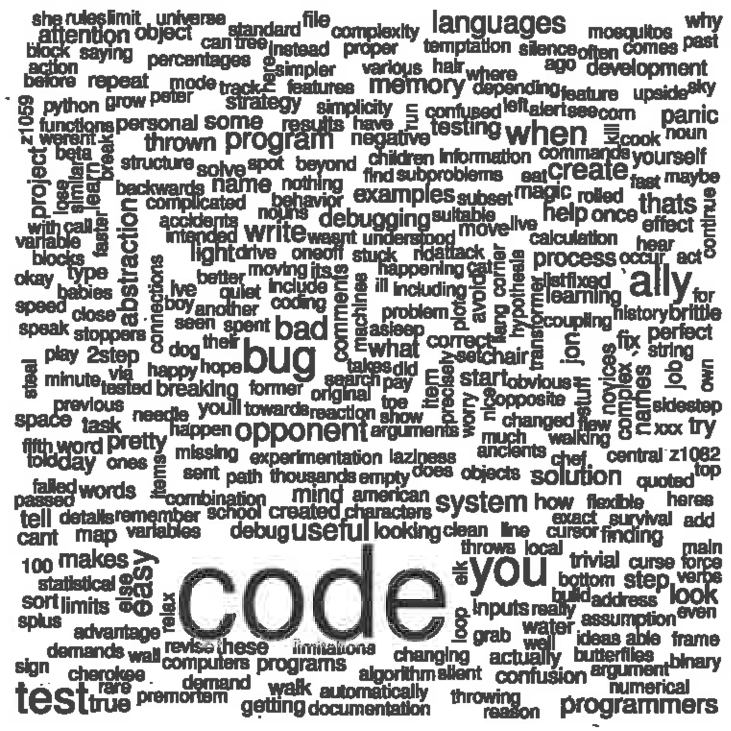
Opponent

- Chapter 29: Be Consistent

Ally

- Chapter 37: Become a Ghost
- Chapter 43: Give Them Their Own





Chapter 30

Relish Magic

Language is magic.

With some arbitrary marks I am right now putting pictures into your head that you've never seen before. By creating a few sounds you can make people imagine something of your choosing.

Programmers are interested in linguistics. I suspect the attraction goes both ways.

Programming tends to come easier to people who have an appreciation of natural languages. That slight advantage will be reinforced because people do more of what comes easiest to them.

Once you've done programming for a while, the similarities to natural languages start to jump out. Natural language becomes more intriguing.

Agreement

Natural languages have magic that computer languages do not. We don't have to completely agree on the definition of the words we use in order to communicate in natural languages.

Not agreeing on definitions in programs is a source of bugs.

This can be an emotional trigger point. Computers demand to have complete control over definitions. Novices are used to being able to negotiate definitions. The computer gets cast as uncooperative.

Let go of control over definitions.

Ally

- Chapter 4: Carve Reality
- Chapter 9: Verbalize and Nounalize

Chapter 31

Tell a Good Story

The names you use have a big impact on how understandable the code is.

Sometimes x is a perfectly good name for an object. Much more often, it is just a lazy name. If the object has a particular rôle or is expected to be a certain type of data, then make that evident. Your code is telling a story, and the story won't be very good if the characters don't have good names.

The names should not be easily confused. If you have two names like xx2 and xx3, change them. No one, including yourself, is going to always keep those two names straight.

Make your names descriptive and confusion-proof.

Overcome trouble before there is trouble

Debugging

One of the most effective ways to speed up debugging is to create a good roster of names. The speed comes because:

- There is less debugging to do
- There is less confusion

Temporary

Do not create a name meaning "temporary".

You don't want characters completely changing their personality in the middle of the story. Two concepts should mean two names.

Surprise is Bad

Names should clearly say what they mean.

Surprise is Good

Names should jar people into recognition.

Story 1

z1059 likes z1039. z1039 is sort of indifferent to z1059. z1082 approaches z1059. z1059 re-

buffs z1082. z1039 sees the interaction between z1059 and z1082, and decides to go for z1059.

Story 2

Nadia likes Liang. Liang is sort of indifferent to Nadia. Juan approaches Nadia. Nadia rebuffs Juan. Liang sees the interaction between Nadia and Juan, and decides to go for Nadia.

Empty Cans

Benjamin Whorf was a linguist, but he had a day job — he did risk assessment for an insurance company. He noted that a lot of fires were caused by "empty" gasoline cans. But "empty" in this case doesn't actually mean empty at all, it means full of gasoline vapor. It is the vapor that is the flammable bit — liquid gasoline doesn't do much of anything except in extreme circumstances. He convinced at least one plant to not allow smoking in a room of "empty" drums.

Speaking of flammable, Whorf was apparently influential in promoting the use of "flammable" rather than "inflammable" since the later can be misconstrued as "nonflammable".

Ally

- Chapter 37: Become a Ghost
- Chapter 39: Do Nothing Well

Chapter 35

Engage Eyes

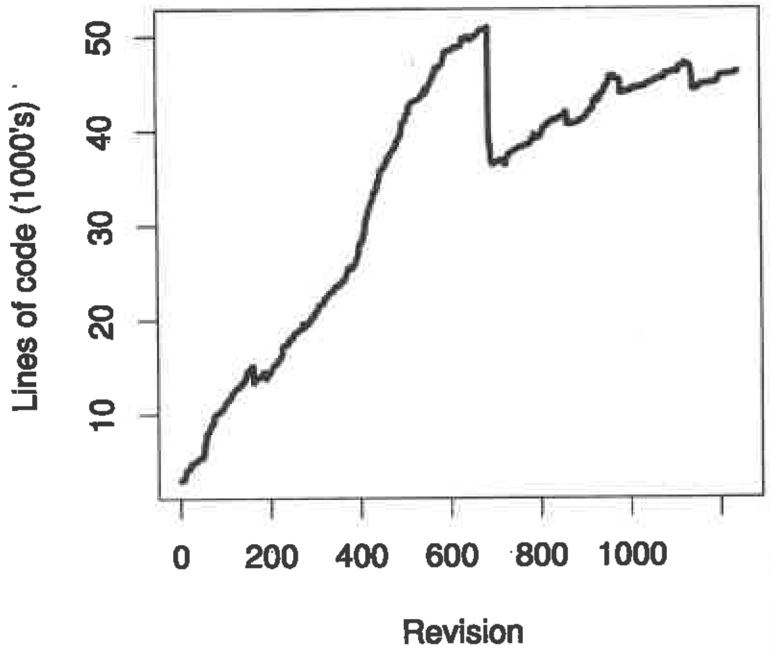
A large portion of the human brain is devoted to vision. Speak to that portion as much as possible in everything you do.

Make things visual:

- for your users
- for you when you plan
- for you when you code
- for you when you debug

Lines of Code

The figure shows the number of lines of code for a project over time.



There was an increase in speed and breadth of functionality over this time period. Novices may think that the increase in the number of lines is good. More experienced programmers will know that the good parts are when the number decreased.

Chapter 36

Grow a Cathedral

The natural state of software is to evolve over time.

Don't build a hangar with rigid dimensions. Cultivate a cathedral that can easily be expanded and reshaped.

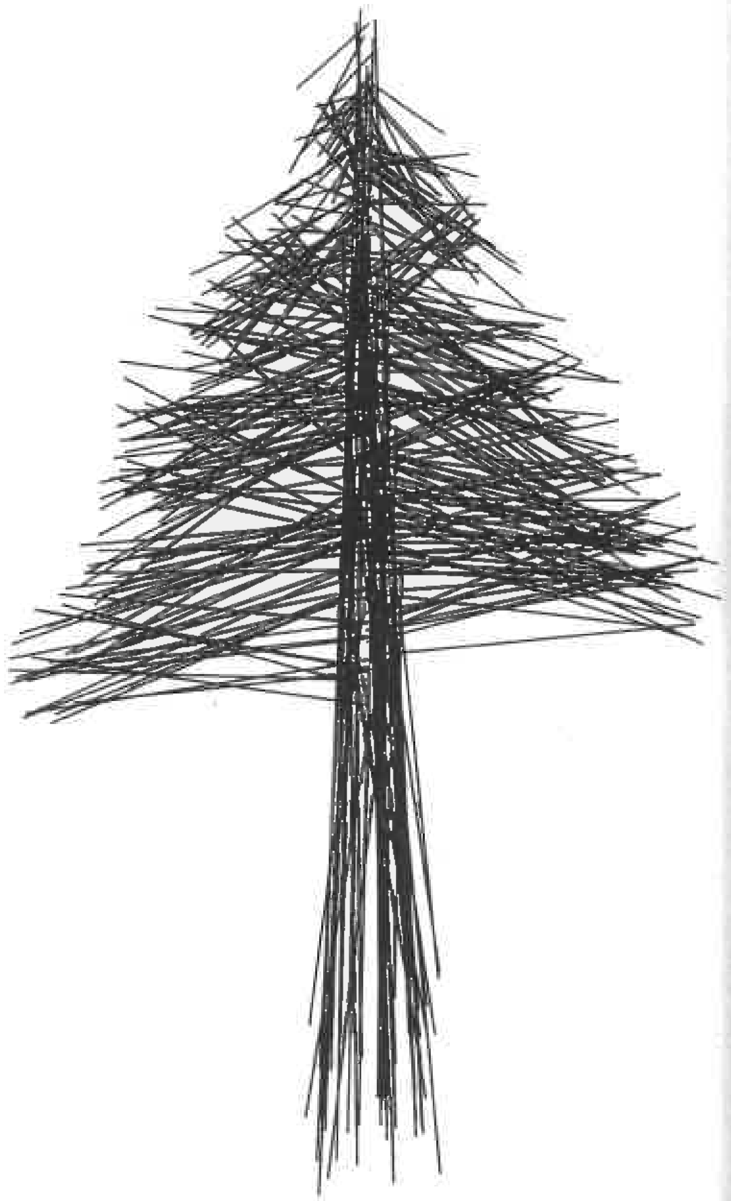
Like a river flowing home to the sea

Logical Arguments

Arguments are often logical values — two values, either true or false. If the argument corresponds to something that really is either true or false, then that is the correct choice. But often that argument can be cast into the context of a menu of possible options, only two of which are currently being implemented.

Make it easy for the code to blossom and grow.

Try to make your code forward compatible.



Chapter 41

Give Up Control

If what you need has already been done (and done well), then use that.

Nut Cracking

A group of crows have nuts that are hard to crack. They fly above an intersection and they drop their nut in the crosswalk. When the traffic light is green in that direction, cars drive over the nut and crack it. When the light changes, they fly down and grab the kernel.

Opponent

- Chapter 44: Don't Borrow, Steal

Veeho Steals from Sun

This is a story from the Cheyenne.

Veeho (the trickster) saw that Sun had some very nice leggings. When Sun was out, Veeho stole them. Sun saw Veeho with the leggings, and asked about it.

Veeho always has a reply for everything. "I just put them under my head so I would sleep well, I knew you wouldn't mind."

Sun knew that Veeho was lying, but played along. "Since you like the leggings, let's pretend that I held a giveaway feast and the leggings were a present to you."

Veeho said, "I was joking, but since you've given them to me, I gladly accept."

Veeho was very excited to get out of Sun's lodge and try out the leggings. Veeho put them on and started running. The grass caught fire, singeing Veeho's legs. Veeho yelled for help, but Sun pretended not to hear. Veeho plunged into a stream, but by then the leggings were ruined.

Veeho asked for another pair. Sun said, "Even I can't make magic leggings but once." Now, Sun could easily have made a new pair, but then Veeho wouldn't have learned anything.

Chapter 45

Always Softcode

Code should make as few assumptions as possible. A common assumption is that some value is a fixed quantity. For example x always has length 5 — that is, the length of x is hardcoded to be 5.

The words "hard" and "brittle" are synonyms. Hardcoding makes the code brittle.

If we want to change the length of x, we'll probably have trouble. It may not be enough to find all the occurrences of where "5" means length of x (which may be a formidable task in itself). Note that changing a 5 that doesn't mean length of x is a wonderful way to create a nicely mysterious bug.

The idea of length 5 may run deep in the structure of the code. So it may be easier to start over entirely rather than try to modify the existing code.

Using a variable to hold the length of x is a case of *indirection*. We are indirectly stating the length.

If there is hardcoding, there is missing abstraction.

Hermit Crabs

Hermit crabs softcode. They find a shell that fits.

Ally

- Chapter 4: Carve Reality
- Chapter 54: Do Not Repeat Repeat Repeat

Chapter 46

Topple Fences

Avoid arbitrary limits.

Limits can be good. Restricting the year of birth of a living person to 4 digits can make everyone's life easier.

But restricting a general year to 4 digits is the Y10K bug (and may rule out the birth year of Lao Tzu).

Really terrible limits are ones that have no basis in the domain reality, only the reality of the system or algorithm that the programmer is working with. Examples are:

- a string holds at most 512 characters
- at most 31 variables are allowed in a problem

Arbitrary limits are ugly. They are a clue that the code is overly full. Be hubric enough to get rid of ugly limits.

Zero One Infinity

The Zero One Infinity Law says that if there is a limit on a number, then that limit should be either zero or one.

Opponent

- Chapter 47: Be Claustrophilic

Ally

- Chapter 28: Have Hubris

Chapter 47

Be Claustrophilic

We all like our freedom, but constraints — even stupid ones — are sometimes our best friends.

What's special about poetry relative to prose? Limitations. Much of the beauty in poetry is in successfully conquering the (self-imposed) constraints.

Some of Billy Wilder's best movie scenes were the result of working around the censorship code.

Imposed limitations are a blessing if they force you to think creatively about the solution.

Yielding is strength

Limitless

If you have no limitations, think some up and consider how that would change the implementation. Two possibilities are significant restrictions on execution time and on memory use. But try for more creative constraints.

Opponent

- Chapter 46: Topple Fences

Ally

- Chapter 23: Curse Knowledge

Chapter 48

Be Wary

The sage is cautious as a fox on thin ice

The first things you should ask when you get the results from any program are:

- Does this make sense?
- Is it reasonable?
- Is there a way to check the results?

(True for non-computer results also, but never mind.)

If you learn to be wary of everyone else's programs, you will be better able to check your own.
— Kernighan and Plauger

Demand

During a negotiation between the English and the French, one of the French messages used the verb *demander*. The English were incensed at the French making demands. The proper translation was not “demand” but “ask”.

Programming languages have “false friends” as well.

Smell

There’s the concept of “code smell”. Code smells if it seems to be too complicated and wordy for the task, if it is ripe for bugs. The person thinking it smells may not see any bugs, perhaps there are no bugs to see, but it can still be a breeding ground for them.

Code that smells is a large, diffuse kludge.

Opponent

- Chapter 49: Lose Every Battle

Ally

- Chapter 50: Avoid the Plague

Chapter 49

Lose Every Battle

Once you believe you have the proper abstraction for some functionality, you need to tell the computer about it — you have to crystallize that abstraction in your chosen computer language.

The translation from your head to the code is unlikely to be perfect.

Novices often imagine that experienced programmers don’t make mistakes. Maybe there are a few, not many I don’t think.

It can be a good strategy to test each few lines of code that you write to make sure they do as you expect. This is easy to do in languages like R, Python and Matlab. It is more bothersome in compiled languages like C++, but can still be done.

When I write a function in R, I often include a line containing “`browser()`” at the end of the function. I’ll write a few lines, and then test it. The call to `browser` allows me to inspect the variables in the function to see if they are as I expect. Often they aren't.

Code under development should chatter like a toddler.

My personal strategy is to win the war by losing every battle. I make lots of mistakes as I code and fix them along the way. It seems to be a pretty good strategy.

But I would say that, wouldn't I? It's not like I have much choice. It's not entirely accidental that I wrote the book (*The R Inferno*) on how to screw up in R.

(I'm especially bad at binary choices. It seems that since there are only two choices, my brain thinks it is always easier to do the experiment than to think it through.)

Opponent

- Chapter 42: Be Quiet
- Chapter 48: Be Wary

Ally

- Chapter 11: Be Accident Prone
- Chapter 60: Know Why It Works

Chapter 50

Avoid the Plague

the plague (noun): The condition of getting a wrong answer with no indication that something is wrong.

This is the key reason that spreadsheets are so dangerous — it is very easy for this to occur with a spreadsheet.

Better the computer should blow up than you unwittingly accept a wrong answer.

It's hard to make a computer blow up on purpose via software (I've tried). The better and easier route is to throw an error if something might go wrong.

Kill To Be Kind

When an error is thrown, then whatever is happening is interrupted and an error message appears. Hopefully the message is comprehensible to mortals.

When a warning is thrown, the computations continue but a warning message appears (somewhere).

Some places put up no-parking signs. Other places build a metal fence so no one can physically park there — a no-parking sign would be redundant.

Warnings are no-parking signs, and errors are fences.

The metaphor breaks down in that there are times when a warning can be safely disregarded. If that is not true of a particular warning, then it should be an error.

Reasonable no-parking signs give times when parking is okay. With warnings it is up to the user to decide in each instance if there is trouble or not. Warning messages should never be ignored, but they can be considered and then discarded.

When you are writing code, you have three possibilities:

- error
- warning
- no problem

If something is not clearly an error, then you have to decide. Considerations should include:

- the probability of a false negative (the computation is bad, but an error is not thrown)
- the likely consequences of a false negative
- the probability of a false positive (the computation is good, but an error is thrown)
- the amount of hassle a false positive causes

The R language provides an example. The condition in an `if` statement should be a single logical value. It would be reasonable for an error to be thrown if the condition contained more than one value. What actually happens is that the first value is used and a warning is thrown.

If the condition has more than one value, then something is definitely wrong. Often it is the logic of the code. I've seen an advantage of getting a warning instead of an error in this case in that it shows me the next bone-headed thing I did when writing the code. I get to fix two things for the price of one.

But it need not be the code that is wrong, it can be that the data going into the code is an unexpected size. My experience with this case is that the answer is often correct, and I merely vow to be more careful in future.

The danger of not throwing an error is that warnings are often ignored.

Never a Warning

There is a radical theory that a user should never see a warning.

I disagree with the theory. And I quite like the theory.

I agree that naive users should never see a warning.

Programmers at intermediate levels should see warnings though. Warnings are for events that are ambiguous — sometimes the event could be bad, sometimes it's okay. If you create a program for naive users, then the program should be specific enough to make clear dividing lines between errors and no problem.

This is easy if your computing environment has the capability of both promoting warnings to errors, and suppressing warnings.

Write Informative Messages

In the Dark Ages error messages used to be something like:

```
Error 43854
```

How informative.

Many modern error messages are not much better. A message like:

```
Bad value for x
```

fails to help the user not to panic. There is no invitation to keep thought processes flowing. A better error message would be something like:

```
'dollarValue' expected to be
numeric of length 1
-- given is character of length 29
```

The added information may clue the user in to what has gone wrong. At the very least it gives them something to think about other than that they have hit a roadblock.

Stealing Light

The Cherokee tell this story.

There was only darkness in the beginning. Fox knew of some people on the far side of the world who had light, but they were too selfish to share it. Possum went to steal the light. Possum hid it in her bushy tail, but the light burned the tail hair off and Possum was caught. Next Buzzard said, "I have a better plan, I'll hide the light on my head." So Buzzard went off to steal light, but the light burnt the head feathers. The other people got the light back from Buzzard as well. That is why possums have bare tails and buzzards have bare heads. Grandmother Spider succeeded in stealing light.

new handles and 3 new heads since then) in that someone along the way has put some thought into the function.

Code written "just for this one time" may live way beyond its first use and become entrenched in some system.

The moral of the story is, I think, to always code as best you can. Also be wary of code that comes your way via inheritance.

Ally

- Chapter 24: Rise to the Occasion

Chapter 53

Beware Easy

Easy is as easy does.

Number to Character

One day when I was an S-PLUS developer I decided that the function that formatted numbers into strings needed to be beefed up. So with a dose of good intentions and some Jamaican Bobsled Theory (how hard can it be?), I set to work.

It has to be easy, right? I'd been writing numbers at least since I was 6. Well, it turns out that something really simple can be incredibly hard. Months later I still wasn't convinced that we had found all of the bugs.

Programming numerical problems is actually relatively easy — we expect it to be hard because the mathematics is hard for us. But many of the things we take for granted are hard to express in programming languages.

Binary Search

Binary search is easy. If you are looking for a specific item in an ordered object (like a dictionary), just see what the item is that is half way between the ends of where you know your item must be. Either the new item you just picked is what you are looking for or you have just halved the interval in which you know it resides.

Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.
— Jon Bentley

Opponent

- Chapter 28: Have Hubris

Chapter 54

Do Not Repeat Repeat Repeat

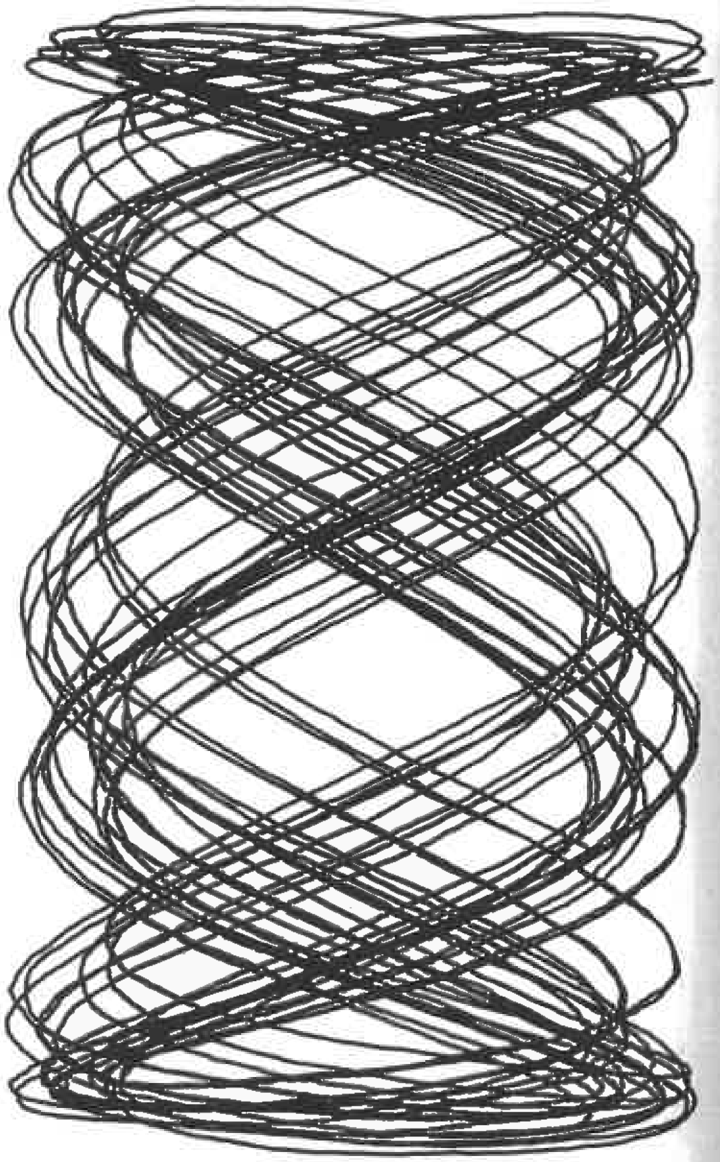
If there is repetition in your code, then you have missed an opportunity for abstraction.

But repetition is not just an opportunity lost, it is a positive danger.

If you want to make a change to the repetitious code, then you have to change every occurrence. And before you can change them, you have to find them.

If the change is a bug fix and you miss one, then you have just created a pernicious, hard-to-find bug. ("I already fixed that bug, it has to be somewhere else.")

If the change is a modification to the behavior and you miss one, then you have just created a pernicious, hard-to-find buglet. Buglets can grow into bugs.



Chapter 58

Understand Bugs

I had a mystery — for years. It was not a mystery I pondered. I classified it as curious and tossed it aside.

The mystery was that the end result of a debugging event would very often be fixing two or three bugs, not one.

Then one day I understood. As Edwin Land would say, I had a sudden cessation of stupidity. It was not at all coincidence that I often found multiple bugs in one go.

There are two types of bug.

Shallow bugs

There are trivial bugs. Typos, flubs. The things that novice programmers think make a program good when they are absent.

These are what unit tests are for.

They are annoying when they appear, but they are easily dealt with.

Deep bugs

Deep bugs are different. They lie silent, possibly for years. They only bite in rare circumstances.

The piece that I was missing in regard to finding multiple bugs is the concept of a "normal accident".

Our common sense idea of an accident is that it has a single cause — someone got drunk and drove off the road. The reality is that accidents normally have multiple causes. Accidents are rare because the causes seldom align.

A classic example of an accident is the Three Mile Island nuclear plant in 1979. The accident happened along the lines of:

- a filter got blocked
- engineers trying to unblock the filter spilled a fraction of a liter of water
- this caused the safety system to automatically shut down the pumps circulating water through the reactor
- two backup pumps should have kicked in but their valves were mistakenly left closed after maintenance

- an indicator light in the control room was hidden by a repair tag
- so an operator thought there was too much coolant rather than not enough and did an override
- a pressure relief valve opened as it should have when the pressure was high, but stuck open
- the control-room indicator of the relief valve was not working

Any one of those things not happening, and nothing to not-much would have gone wrong, maybe.

In my debugging sessions I was seeing normal accidents. Two or three bugs, harmless in their own right, would form a conspiracy.

Lesson

If you seem to have a deep bug, look around for all of the contributing causes. Don't just stop with the first thing that makes the bad behavior go away.

Systems

Normal accidents occur when:

- the system is complex

- there is tight coupling

How many programs aren't complex with tightly coupled pieces?

Our motto of simplicity is looking good here. The surface solution to tight coupling is modular programming. But watch out — coupling can go deeper than the modules.

Engineers have seen that safety systems often make things worse — notice how many of the items in the Three Mile Island incident were done in the name of safety. Better to simplify the problem away rather than try to fix it with complex additions.

Ally

- Chapter 59: Spot Bugs
- Chapter 63: Dance the Debug 2-Step

Chapter 59

Spot Bugs

My wife and I were walking through the woods with some children. One of them said, "There's a frog."

"Where?"

"Right there, on that tree."

After a minute or two of peering at the tree from half a meter away with prompts regarding the exact location, we finally saw what he had effortlessly seen from three meters.

That boy saw frogs. I see bugs.

I'm a pretty good programmer, but I'm not a Great programmer. A talent that I do seem to be exceptional at is spotting bugs. We know that it would be ridiculous to ask that boy how he spotted the frog. (The only semi-feasible idea I know is that his Native American heritage put him closer to ancestors for which such a talent would have a survival advantage.)

A prerequisite for him seeing frogs is for his eyes to be open. I know a few things that seem to be prerequisites for spotting bugs.

- know what to expect
- be relaxed
- be wary as a cat
- be curious (as a cat)

If you don't know the right answer, you're not going to spot a wrong answer. As good as my bug-spotting skills are, I have no chance in an unfamiliar field.

You may think there is a contradiction between being relaxed and being wary. There is. And there isn't.

I suspect that frog-boy satisfied all of these criteria as well. That he was alert, but on wide focus. That he saw the frog peripherally.

History

There is skill and luck. Finding the bug I'm about to describe was pure luck, especially in the circumstances. Still, it seemed to boost my bug-finding reputation.

We were having a development meeting for S-PLUS. It was announced that we were in code-freeze for the upcoming release.

The purpose of a code-freeze is to make sure that what the customers get has been thoroughly tested. The temptation is to fix every bug found until it is time to ship. Everyone who's been around a while learns that fixing bugs is a magnificently efficient mechanism for creating bugs. People have learned to do code-freezes.

The only bugs that would be exempt from the code-freeze were system-terminating bugs. A system termination occurs when it is detected that objects could become corrupted. The responsible thing to do is for the system to shut itself down. In biology this is called apoptosis. Suicide for the greater good.

System-terminating bugs are the most serious. It is worth the risk of adding new bugs to get rid of a really serious one.

Of course this means that it isn't really a code-freeze, it's a code-slush. The word that is used for a real code-freeze is "release".

The development manager announced that he would buy a drink for the first person that found a system-terminating bug.

The meeting broke up. I went to my desk. I typed a line, got a system termination. Within a half hour of the end of the meeting I had a bug report filed.

I don't remember the preliminaries to trigger the bug, but I do remember the punch line. The command was:

```
    history()
```

As in: type this and S-PLUS is history.

My drink of choice is sparkling water so the development manager got off light.

Ally

- Chapter 58: Understand Bugs
- Chapter 63: Dance the Debug 2-Step

Chapter 60

Know Why It Works

There are two types of code:

- code with a clear logic
- code that seems to work

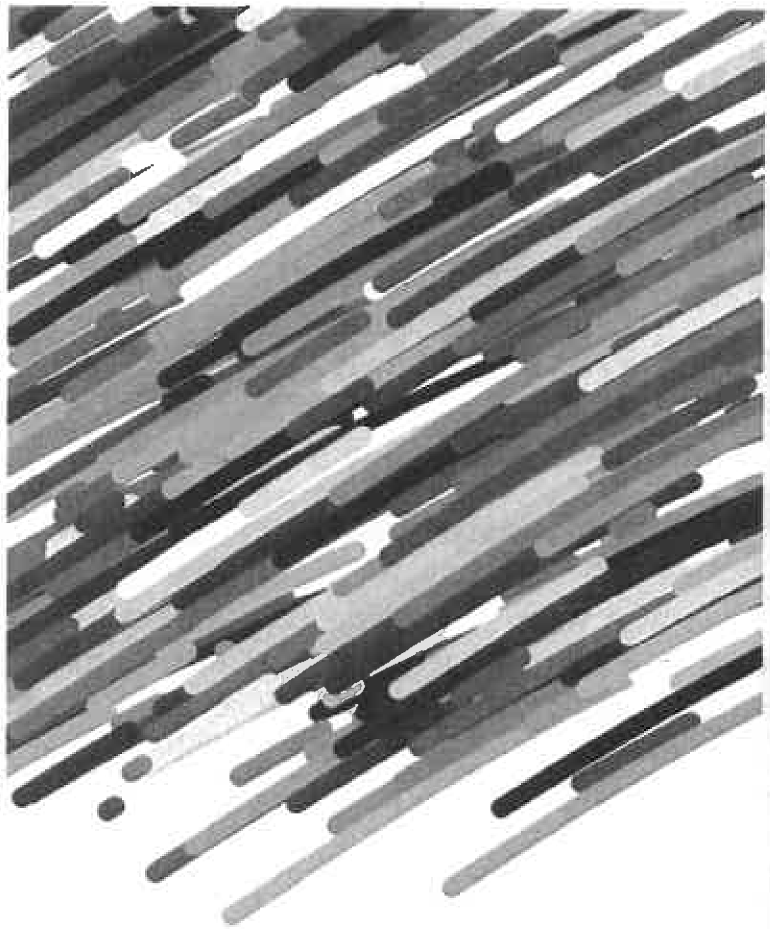
When you write some code and it doesn't work (as per usual), the temptation is to experiment with changes to the code until it seems to work. That's not programming, that's gambling. That's the devil talking.

What you should do is understand **why** it is not working, and formulate the logic that will make it work.

Furious activity is no substitute for understanding.
— H. H. Williams quoted by Jon Bentley

Ally

- Chapter 61: Think Safety



Chapter 61

Think Safety

Once upon a time there was a program that automatically sent out a report, including graphics, every day. That's great. That's what computers are all about — doing mundane stuff that we get bored doing. And in the beginning it was good.

Then people moved on. The job wasn't so well-monitored any more. Eventually the subscribers of the report began to take more interest in it. Not because it had suddenly become more informative, but because it had become more entertaining. There was no telling what ridiculous looking plot would appear next.

I'm all for entertainment, but you want to be famous for entertainment that you do purposely. (I'm happy to report that it was not my code driving the process.)

It is essential to provide sanity checks for automatic processes. An automatic process will almost surely get updated data and then perform some processing of the new data. The process itself should ask the questions:

- Did we get all the new data we should expect?
- Does the new data seem believable?
- Are the results sensible?

Sometimes the new data can have an overlap with the data we already have. If the data in the overlap period match, then we can have greater faith that the new data are correct.

Unleashed dogs need to be well-trained.

Better a stride too far back than a toe-length too far forward

Make a Checklist

If your code were an airplane, what checklist would you create to ensure that you didn't crash as you flew across country?

Now, which of those can you make redundant by changing the code?

Flock

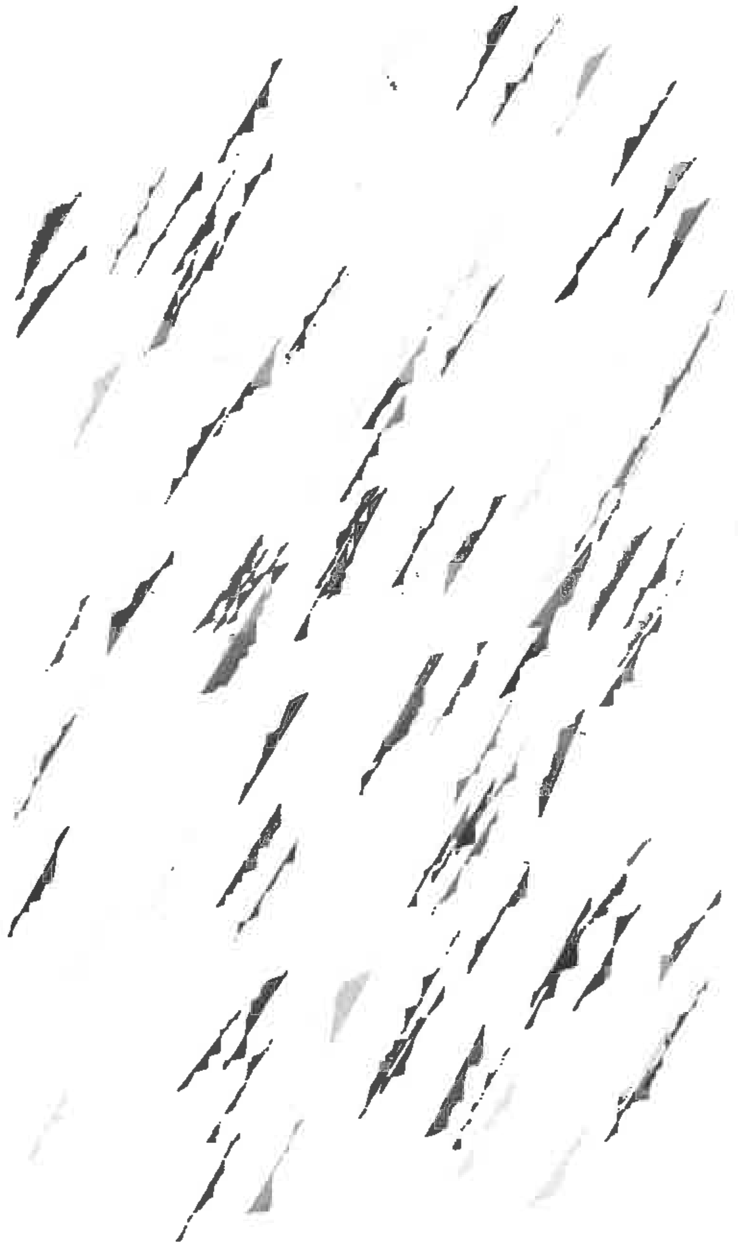
Why don't flocks of birds need big, fancy air traffic control systems? Millions of bird-hours, no collisions.

Each bird keeps track of a few neighbors:

- don't be too close
- don't be too far

Ally

- Chapter 60: Know Why It Works
- Chapter 77: Tattle on Yourself
- Chapter 75: Prove Yourself Wrong



Chapter 64

Clean Up After the Flood

You have found and fixed a bug. You feel good, you think you are done.

Feeling good is appropriate, but you are not done.

Step 2

- Look for other occurrences of the same bug.

Step 3

- What bugs is this bug a special case of? Look for those bugs.

The meaning of "look for" should be taken to mean both look through the code for possible occurrences of the bugs, and design tests to ferret out such bugs.

Ally

- Chapter 63: Dance the Debug 2-Step
- Chapter 75: Prove Yourself Wrong

Chapter 65

Play

That is, build prototypes. Whenever you have a new thing to do, explore what lots of solutions might look like. Children experiment to figure out how their world works. You are a child in the world of your new project.

Dress it up as exploring the solution space if you must.

The sage acts without doing

Really Play

Do interesting stuff for no real reason.

Dozens of times since then I have seen today's toy turn into next week's beast of burden or next year's product.
— Jon Bentley

How the Elephant Got His Trunk

Some people think that an essential ingredient of a good programmer is curiosity. Develop yours.

Ally

- Chapter 18: Be a Hacker
- Chapter 26: Be Lazy
- Chapter 78: Be a Polyglot

Chapter 66

Sidestep Show Stoppers

You pile family, tents, food, clothes and so on into the car. You drive off into the wilderness. You start to experience wonderful fresh smells. You run out of fuel before you reach your destination. Not good.

I've known no one who has done this with a car. I do know people who have done this with programming projects. Let's spend months coding only to come to the realization: "Hey, this isn't gonna work."

If you are embarking on a big project, then the first thing on the agenda should be to look for roadblocks. Can you get from one end to the other? This requires an anti-procrastination strategy — you need to investigate all the hard problems to make sure they really are tractable.

Opponent

- Chapter 8: Procrastinate

Ally

- Chapter 67: Do a Premortem

Chapter 67

Do a Premortem

Do a premortem when you start a major project. This is a game where the supposition is that the project has failed (at some indeterminate time in the future). The object of the game is to state why it is that the project failed.

Our optimism bias is a problem. Suppose you asked, "What is wrong with the project?" The answer would be, "Nothing, it's fine."

But with a premortem we dictate the death of the project. Now it becomes okay — both psychologically and socially — to come up with weaknesses of the project.

Ally

- Chapter 66: Sidestep Show-Stoppers