



Principle of Least Expressiveness

George Fairbanks

I'M ALWAYS DELIGHTED to discover a connection between two ideas that I'm already fond of on their own, so I'd like to share a connection I found recently. The first idea is writing code that expresses my thinking about the problem domain, and the second is the principle of least expressiveness (PLE). The connection is that I can use the PLE to reveal my thinking about the problem domain, and because all ambiguity stops at the code, the act of programming using the PLE can help me simplify and debug the flawed ideas I have in my head.

The PLE³ is as follows:

When programming a component, the right computation model for the component is the least expressive model that results in a natural program.

The least expressive model means that if you can express your idea with a constant, use that, and similarly for lookup tables, state machines, and so on. You should only use a Turing-complete language when you cannot use something simpler—with the caveat not to contort the code.

We can see the same idea with a different name in the Rule of Least

```
if (file.format() == JPG
    || file.format() == PNG) {
    return new DecoderA().decode(file);
} else if (file.format() == GIF) {
    return new DecoderB().decode(file);
} else {
    throw new IllegalStateException();
}
```

FIGURE 1. Using IF/ELSE to invoke the right decoder plus express the relationship between formats and decoders.

Power, where it guides the architecture of the web. Berners-Lee and Mendelsohn¹ put it this way:

The Rule of Least Power suggests choosing the least powerful language suitable for a given purpose. ... If, for example, some weather data is published as a Web resource using RDF [Resource Description Framework], a user can retrieve it as a table, perhaps average it, plot it, or deduce things from it in combination with other information. ... The only way to find out what a Java [weather] applet means is generally to set it running, and see what it does.

The PLE is like many other design principles in that you may have discovered it independently, already use it when you write code, and yet still

find plenty of other code violating it. If nothing else, perhaps this article connects your good design instincts with a named concept and you can point to it during code reviews or mentoring. For those who haven't already been down this path, let's work through how using the PLE can improve your code.

Saying More by Expressing Less

Imagine that our program has a fairly common task: to decode files in various formats. Not knowing any more about the problem, we can guess that one of the following is probably true, but which one?

1. one format and one decoder
2. one format and many decoders
3. many formats and one decoder
4. many formats and many decoders.

Figuring out this relationship before writing code is instinctive to anyone who has worked with database schemas or any careful software design process. How might this relationship reveal itself in code? Figure 1 shows a typical way to use an IF/ELSE statement to implement decoding. A CASE statement would look quite similar.

From this, we can see that a format maps to one decoder and a decoder maps to many formats, but we

have to look around a bit to decide that. How well does this code rate from the perspective of the PLE? Well, it uses a Turing-complete language to express a relationship that's tabular, which is more than necessary. As shown in Figure 2, we can express the table directly in the code, at which point the rest of the code shrinks to just a lookup. What's more, it no longer takes any effort to see what the relationship between formats and decoders is.

Readers can look at this code and think, "It doesn't look like this mapping ever changes." That's right, and it's easy enough for us to say that more clearly. Figure 3 shows the same map but declares it as final and uses the Guava ImmutableMap class, making it unchangeable at runtime. I call this *nailing it down* because things that are nailed down don't move and don't complicate my thinking. I nail down as much as possible so I have fewer things to think about.

By the way, you might notice that the code in Figure 3 has only one semicolon. I have never really paid much attention to Java's design decision in the early collection classes (e.g., HashMap, as seen here) to use statements instead of expressions. However, that decision makes it impossible to declare a new HashMap and populate it with data at the same time. In contrast, the ImmutableMap in Figure 3 can be declared and populated in a single expression.

I've noticed that, over the code's life, people will make expedient edits to it and won't always pause to think whether the edit belongs there or elsewhere. The original IF/ELSE variant lends itself to expedient inline edits because it's so easy to just add a new line within the curly brackets. The revised code

```
Map<Format, Decoder> decoders = new HashMap<>();
map.add(JPG, new DecoderA());
map.add(PNG, new DecoderA());
map.add(GIF, new DecoderB());

return getOptional(decoders, file.format())
    .map(Decoder::decode)
    .orElseThrow(IllegalStateException::new);
```

FIGURE 2. Expressing the relationship between formats and decoders as a table, separated from the decoding and exception handling.

with tabular data does not lend itself to that kind of edit, so it might maintain its design integrity better over time.

Which version of the code you prefer depends partly on what kind of code you are used to seeing. Once you are past that, however, I think the revised version is the clear winner. It has better separation of concerns, with one part tabularizing the formats and decoders and the other part doing the decoding and exception handling. Adding a new format and decoder pair is localized. If we're lucky, the decoder lookup lines will never need to change.

The revised version is also easier to read and reason about. When I'm reading the declaration of the map, I'm just paying attention to the entries themselves, not reasoning through a Turing-complete language. This is the key benefit of the PLE: reading that part of the code doesn't require me to mentally simulate a complicated language. It's both less mentally taxing and less prone to error.

Consider doing a code review on Figure 1 versus Figure 3, especially a scaled-up version with lots of cases. If there were bugs in the logic, I think I'd be less likely to catch them in the IF/ELSE style simply because

```
final Map<Format, Decoder> DECODERS =
    ImmutableMap.builder()
        .add(JPG, new DecoderA())
        .add(PNG, new DecoderA())
        .add(GIF, new DecoderB())
        .build();
...
```

FIGURE 3. The mapping between formats and decoders does not change at runtime, so we can express it as immutable.

my brain has to work harder to follow the logic. I've found myself many times becoming numb during reviews when the code is intricate but repetitive. Presented as a table, it feels less tedious to scan.

Reveal Your Thoughts About the Problem

Any time you revise your code down from a Turing-complete language to something simpler, you're not giving your thoughts anywhere to hide. If you show a state machine with exactly one transition from A to B, that's not just an implementation choice—you are saying that you've thought about the problem, that there are two states worth paying attention to, and

that it's possible to go from one to the other but not the reverse.

The act of choosing a less-than-Turing way to express the problem requires you to be more precise. It's not just a variable that doesn't seem to change—it's a constant. It's not just a table—it's a bimap. It's not just an update operation—it's a discrete state transition.

When you read code that's written using the PLE, you are seeing the problem domain through the eyes of the code author who acts as a tour guide, telling you what is worth paying attention to and how it behaves.

Activate Your Critical Thinking

The idea is that by applying the PLE, we can write clearer code that better reveals how we think about the problem. But wait, surely we can't explain something more clearly than we understand it nor can our code be clearer than our thoughts. A mechanical application of the PLE cannot create simplicity from a tangled understanding.

Here's the trick: revealing your thoughts in the unforgiving environment of code has the delightful effect of activating your critical thinking to focus your fuzzy thoughts. When they are in your head, your thoughts are not subject to a compiler, type checker, or regression tests; but in code, they are. This is true even without using the PLE.

Using the PLE leads you to poke and prod the idea more thoroughly and from new angles. Consider our format and decoder example again. Once the relationship was expressed as a table, perhaps you wondered if the relationship also works in the reverse direction. Perhaps you thought about what should happen if more than one decoder works on

a format—should that be expressed in our table and the choice among the suitable decoders pushed into the lookup code? And if your math classes stuck with you, perhaps you even considered if the relationship was injective, surjective, or bijective.

These kinds of questions naturally arise from the way the idea is structured—in this case, as a table—and may not arise in other representations. When I read the IF/ELSE code, I only seem to activate the parts of my brain that ask, “will this work?” and not, “what is the nature of formats and decoders?” The closer I get to expressing that nature directly, the more I trigger my brain to ask whether I've got it right or not.

When you structure your code as a state machine, you reasonably ask what the legal state transitions are. In contrast, it doesn't make sense to ask about legal state transitions in general Turing-complete code on general data structures, even if that code has identical behavior. It's the act of casting your thoughts into the less expressive representation that stimulates a cascade of reasoning that shakes out an improved understanding. As is so often true, Fred Brooks talked about this a long time ago²:

Much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies. Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

By using the PLE, we reveal our thoughts to readers in the least

complicated way. But the connection that makes me so delighted is the way it leads to exactly the kind of critical thinking that hones our thoughts.

It's one thing to resolve to think more clearly and quite another to achieve it. My experience has been that, both individually and as a mentor, I've been able to simplify programs to reveal simplicity by applying the PLE mechanically and then asking questions about the resulting program. If nothing else, I've factored out the parts of the program that were constants in disguise, leaving the interesting parts of the code in plainer view. But more often, it has led to insights because what I was manipulating is no longer obscured by too-powerful language.

Clarity Through Definitions

I find a lot of software developers write code and comments that are blandly noncommittal about their precise thoughts about the problem domain. As we all know from posting questions on the Internet, the best way to get constructive feedback isn't to say something bland but to be not quite right, at which point people will rise from the dead to correct your mistake. If constructive criticism is what you want (and it is), then you want to be as clear as possible to trigger exactly that reaction.

Imagine a program expressed in a Turing-complete language that adds up some areas and then, ta-da, returns a number that it claims is the total area. Hmm, it sounds reasonable because we have some intuition that the total area must be the sum of some smaller areas. No alarm bells go off, no code reviewer objects, and that code goes into production.

Contrast that with code that states a definition of total area—for example,

that it is length times width. That too seems reasonable, but then a reviewer chimes in and asks, “Are you assuming that the areas are always rectangular? Because in this domain sometimes the angles aren’t quite 90°, so that definition doesn’t always work.” Instead of sliding into production, the misunderstanding is caught, the program is revised to express the proper definition, and anyone reading the code will avoid making the same mistake.

To me, a corollary of the PLE is seeking out places in the code where I could have used vague Turing-complete language and instead used equations or other kinds of definitions. Definitions are incredibly terse and usually falsifiable. So again, your ideas have nowhere to hide.

Time is the best teacher, but it kills all of its pupils. It may sound scary to leave your ideas exposed, but it is the best way to grow as a software designer. I remember the years I spent thinking through designs with invariants and precise preconditions and postconditions. That time was well spent, but it was a personal journey, not a nugget of understanding that I can hand to someone else and have them get to the same place faster than I could.

The PLE, in contrast, is one of those nuggets. It guides you to write code that, on face value, is easier to read and understand and, by expressing the code in a less-than-Turning form, activates targeted critical thinking that results in thoughts that are actually clearer than what you started with.


It’s often remarked that the sign of a good developer is not building something complicated but being able to build something simple. The guidance



ABOUT THE AUTHOR



GEORGE FAIRBANKS is a software engineer at Google. Contact him at gf@georgefairbanks.com.

of the PLE is helpful precisely because you can apply it directly, and as you think more clearly, you find more opportunities to apply it. 

References

1. T. Berners-Lee and N. Mendelsohn, “The rule of least power,” 2006.

[Online]. <https://www.w3.org/2001/tag/doc/leastPower.html>

2. F. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1995.
3. P. VanRoy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. Cambridge, MA: MIT Press, 2004.

Call for Articles



IEEE Software seeks practical, readable articles that will appeal to experts and nonexperts alike. The magazine aims to deliver reliable information to software developers and managers to help them stay on top of rapid technology change. Submissions must be original and no more than 4,700 words, including 250 words for each table and figure.

**IEEE
Software**

Author guidelines:
www.computer.org/software/author
Further details: software@computer.org
www.computer.org/software

Digital Object Identifier 10.1109/MS.2019.2906534