Be aware of the Makefile effect

Jan 10, 2025 **Tags: programming**

I'm not aware of a *perfect*[1] term for this, so I'm making one up: the Makefile effect[2].

The Makefile effect boils down to this:

> Tools of a certain complexity or routine unfamiliarity are not run *de novo*, but are instead copy-pasted and tweaked from previous known-good examples.

You see this effect frequently with engineers of all stripes and skill/experience levels, with Make being a common example[3]:

1. A task (one of a common shape) needs completing. A very similar (or even identical) task has been done before.
2. Make (or another tool susceptible to this effect) is the correct or "best" (given expedience, path dependencies, whatever) tool for the task.
3. Instead of writing a `Makefile`, the engineer copies a previous (sometimes very large and complicated[4]) `Makefile` from a previous instance of the task and tweaks it until it works in the new context.

On one level, this is a perfectly good (even ideal) *engineering* response at the *point of solution*: applying a working example is often the parsimonious thing to do, and runs a lesser (in theory) risk of introducing bugs, since most of the work is unchanged.

However, at the *point of design*, this suggests a tool design (or tool *application*[5]) that is *flawed*: the tool (or system) is too complicated (or annoying) to use from scratch. Instead of using it to solve a problem from scratch, users repeatedly copy a known-good solution and accrete changes over time.

Once you notice it, you start to see this pattern all over the place. Beyond Make:

- CI/CD configurations like GitHub Actions and GitLab CI/CD, where users copy their YAML spaghetti from the *last* working setup and tweak it (often with repeated re-runs) until it works again;

- Linter and formatter configurations, where a basic set of rules gets copied between projects and strengthened/loosened as needed for local conditions;
- Build systems themselves, where everything non-trivial begins to resemble the previous build system.

Does this matter?

In many cases, perhaps not. However, I think it's worth thinking about, especially when designing tools and systems:

- Tools and systems that enable this pattern often have less-than-ideal diagnostics or debugging support: the user has to run the tool repeatedly, often with long delays, to get back relatively small amounts of information. Think about CI/CD setups, where users diagnose their copy-pasted CI/CD by doing print-style debugging *over the network with a layer of intermediating VM orchestration.* Ridiculous!

- Tools that enable this pattern often *discourage broad learning*: a few mavens know the tool well enough to configure it, and others copy it with *just* enough knowledge to do targeted tweaks. This is sometimes inevitable, but often not: dependency graphs are an inherent complexity of build systems, but remembering the difference between `$<` and `$^` in Make is not.

- Tools that enable this pattern are *harder to use securely*: security actions typically require deep knowledge of the *why* behind a piece of behavior. Systems that are subject to the Makefile effect are also often ones that enable confusion between code and data (or any kind of in-band signalling more generally), in large part because functional solutions are not always secure ones. Consider, for example, about template injection in GitHub Actions.

In general, I think well-designed tools (and systems) should aim to minimize this effect. This can be hard to do in a fully general manner, but some things I think about when designing a new tool:

- Does it *need* to be configurable?
- Does it *need* syntax of its own?
    - As a corollary: can it *reuse* familiar syntax or idioms from other tools/CLIs?
- Do *I* end up copy-pasting my use of it around? If so, are *others* likely to do the same?

---

1. The Makefile effect resembles other phenomena, like cargo culting, normalization of deviance, "write-only language," &c. I'll argue in this post that it's a *little* different from each of these, insofar as it's not *inherently* ineffective or bad *and* concerns the outcome of specific *designs*. ↩

2. Also note: the title is "be aware," not "beware." The Makefile effect is not inherently bad! It's something to *be aware of* when designing tools and systems. ↩

3. Make is *just* an example, and not a universal one: different groups of people master different tools. The larger observation is that there are classes of tools/systems that

are (more) susceptible to this, and classes that are (relatively) less susceptible to it. ↩

4. I've heard people joke about their "heritage" `Makefiles`, i.e. `Makefiles` that were passed down to them by senior engineers, professors, &c. The implication is that these forebearers *also* inherited the `Makefile`, and have been passing it down with small tweaks since time immemorial. ↩

5. Complex tools are a necessity; they can't always be avoided. However, the occurrence of the Makefile effect in a *simple application* suggests that the tool is too complicated for that application. ↩

Previously