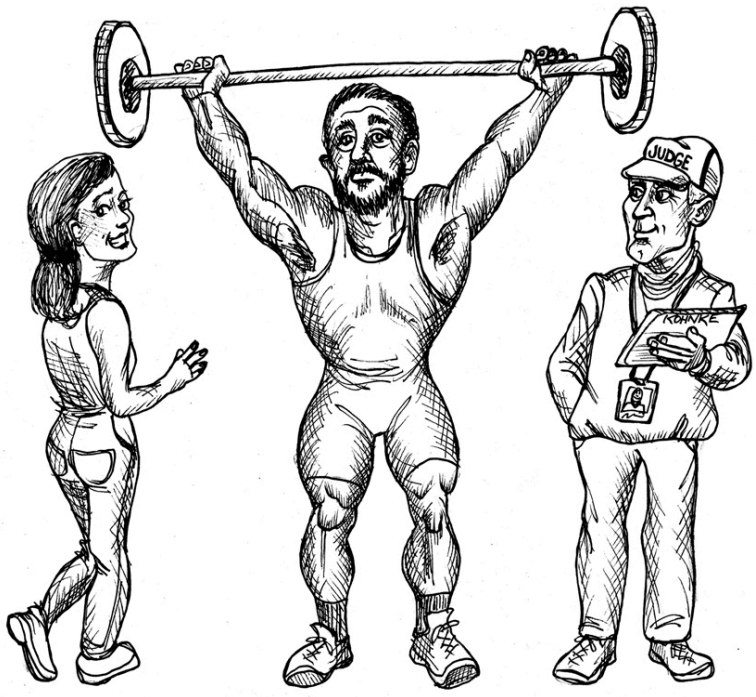
23

The Two Values of Software



There are two values to software. There is the value of its behavior, and there is the value of its structure. The second of these is the greater of the two because it is this value that makes software *soft*.

Software was invented because we needed a way to quickly and easily change the behavior of our machines. But that flexibility depends critically on the shape of the system, the arrangement of its components, and the way those components are interconnected.

Keeping Options Open

The way you keep software soft is to leave as many options open as possible, for as long as possible. What are the options that we need to leave open? *They are the details that don't matter*.

All software systems can be decomposed into two major elements: policy and detail. The policy element embodies all the business rules and

procedures. The policy is where the true value of the system lives.

The details are those things that are necessary to enable humans, other systems, and programmers to communicate with the policy, but do not impact the behavior of the policy at all. These include IO devices, databases, Web systems, servers, frameworks, and communication protocols, among others.

The goal of architecture is to imbue the system with a shape that recognizes policy as the most essential element of the system while making the details irrelevant to that policy. This allows decisions about those details to be *delayed* and *deferred*.

For example:

- It is not necessary to choose a database system in the early days of development, because the high-level policy should not care what kind of database will be used. Indeed, if you are careful, the high-level policy will not care if the database is relational, distributed, hierarchical, or just plain flat files.
- It is not necessary to choose a Web server early in development, because the high-level policy should not know that it is being delivered over the Web. If the high-level policy is unaware of HTML, CSS, AJAX, JSP, JSF, or any of the rest of the alphabet soup of web development, then you don't need to decide which Web system to use until much later in the project. Indeed, *you don't even have to decide if the system will be delivered over the Web*.
- It is not necessary to adopt REST early in development, because the high-level policy should be agnostic about the interface to the outside world. Nor is it necessary to adopt a microservices framework, or an SOA framework, or REACT … Again, the high-level policies should not care about these things.
- It is not necessary to adopt a dependency injection framework early in development, because the high-level policy should not care how dependencies are resolved.

I think you get the point. If you can develop the high-level policy without committing to the details that surround it, you can delay and defer decisions about those details for a long time. And the longer you wait to make those

decisions, the more information you have with which to make them properly.

This also leaves you the option to try different experiments. If you have a portion of the high-level policy working and it is agnostic about the database, you could try connecting it to several different databases in order to check applicability and performance. The same is true with Web systems, Web frameworks, or even the Web itself.

The longer you leave options open, the more experiments you can run, the more things you can try, and the more information you will have when those decisions can no longer be deferred.

What if the decisions have already been made by someone else? What if your company has made a commitment to a certain database, or a certain Web server, or a certain framework? I suggest that you pretend that those decisions have *not* been made, and give the system a shape that allows those decisions to be deferred for as long as possible. You never know when decisions like that might suddenly change.

A good architecture maximizes the number of decisions **not** *made*.