Specification-driven design

Rough transcript of the talk “Specification-driven design” by Joachim Breitner at BobKonf 2022.

Welcome to this BobKonf talk, and thanks to the organizers to having me.

Before I start, let me manage expectations a bit. The following talk is anecdotal by nature: I made an experience that I think is worth sharing together with the ideas behind them. This *is* practical experience from a sizeable production project, so not *just* an idea. But it isn’t a complete fleshed out philosophy that can be applied directly and as it to your project. And I do not claim novelty. Quite contrary, I expect that what I am describing here is more or less an instance of some existing approach out there. That said, I am optimistic that you’ll get some inspiring ideas out of this.

The setting

So here is the setting: Imagine a start-up building something new and novel. An innovative platform or amazing service that the world has never seen before. Or maybe you are just reinventing the wheel again, but your top-notch PR still turns it into a hype. Very likely you have many great ideas about how *your* service works differently, to be faster, more secure, easier to use, more decentralized, or all of these at the same time. You gather a bunch of 10-X programmers around you, all experts in whatever ingredients your secret sauce has, and start building the thing, piece by piece, from the inside out, adding features as you go, until the actual implementations fully fills your vision, and you ship the thing. Great, well done!

But what might happen is that the outer edge of the implementation ends up defining the external, public interface of your service. And thus, possibly, likely, the shape of this externally visible surface happens to be formed around the *internals* of the service. Names, types, protocols are what they ended up after what had to be done to build the thing. Your users might now have to learn more about the *how* of your system than they want to, to make sense of the interfaces. The behavior of the system is mostly defined by its implementation, so it may be neigh to impossible to have an independent, but compatible implementation, for example for n-versioning a decentralized system, or to have special development instances. And even changes to the existing implementation become hard, as it is unclear which changes are ok, and which would be breaking expectations of your customers. Clearly, all in all not so satisfying.

At my previous job we were at risk of doing exactly that. We were building what may be best be described as a decentralized, blockchain-based cloud with a novel programming model based, where services are perpetually running WebAssembly modules communicating with asynchronous messages. And indeed, the system implementation had the ability to pass messages between services before we even had an inkling of how services are installed in the first place, or how they’d send and receive messages between each other and the external world, and which behaviours the users would expect here. We were at risk of treating the public interface like as an afterthought of the implementation, of building the system without knowing whether it makes sense and is usable end-to-end, and without a clear idea of what our system *is*, in an abstract sense.

Furthermore, we had people who’s job it was to build on top of the thing we are building. Example services, tooling, even a complete new programming language that compiles to this new platform. They of course need to know the interface, how it works, which feature is already available and what comes next. They might actually have useful opinions about how a certain feature should be exposed. And ideally they want to start adopting new features as soon as they are agreed upon, and not wait for them to be actually implemented, to reduce the length of critical path.

Specification

At this point I started make a fuzz and annoy the relevant persons enough to let me go ahead with this idea of a “public specification”. The idea is to think of your system from the *outside*. Forget, for a moment, all your clever implementation tricks, and focus on the *what*, not the *how*:

What do your users need to know about your system? Surely, the *interface* of your system. For a network service, this might be the wire protocol. For a web service, the REST endpoints. For a library, it would be the types and functions provided. But the interface is not enough; your users also need to know what these function do, what their behaviour, their meaning is. How they interact with each other, and over time. Or, put more fancily, what their *semantics* is.

So how do we describe the semantics? Isn't this already the "how"-question, the question best answered by implementation? Well, that is one way to define the semantics of your system, but then you have an *implementation-defined* system. But surely your implementation deals with details that you don't want to burden your users with. So we can try to do better and describe only the user-relevant part of the behavior; the essence of your service. Its *specification*.

There are many different forms a specification can take. It could be a bunch of prose describing the intent behind each function or entry point. Or it could be more formal, describing the behavior with mathematical precision (or at least some approximation thereof), starting with a model of the abstract state of your system or service, and then describing for each entry point how the state changes accordingly.

Such a specification document is, in principle, everything a user of your system should ever need. Everything they need to know about it should be written there (or referenced, for example if it is an existing standard). Conversely, if some aspect of your system is meant to be internal, to be an implementation detail, it should *not* appear in this document.

After I finally nagged enough and got some encouragement, I set out and wrote that specification for our system. It was initally called “public spec”, not because the document was public, but because it describes what the users need to know about the system. It has been published since, but is now called “interface spec”, although that name can be misleading, as it contains not just the interface, but also the semantics.

I described the behavior twice. There is a prose-oriented section going through all the various interface, giving their signature where applicable, and describing with words what they do, as you can see here: (show `ic0.certified_data_set`)

But then there is also a more rigorous mathematical modelling. Here I first describe the state of the system, or rather the abstract, conceptual state as it matters to users, with mathematical terms (sets, maps, functions).
(show `S`)
I think this alone makes the work on a spec useful: It really nails down the *what* of the system, carving out this abstract model of the system and separating it from the implementation details.

And then, for all possible interactions with the sytem – handling requests, passing of time, actions of the services hosted on the system – I describe how they affect this abstract state.
(show this)

Such a state machine formalization is quite versatile. For example, it can easily model model non-determinism and other forms of under-specification well. Because likely the specification will not dictacte the behaviour of the system 100%, but leave certain choicse to the implementation. Resources limits, scheduling etc. come to mind. And that is ok, if the spec clearly describes the *range* of possible behaviors and clients (hopefully) don’t rely on more than that.

The document grew quite a lot, as the system grew, and is now almost a book. And certainly not the most accessible document, but I don’t think it has to: It is not a tutorial for beginner or the place to look for design justifications, but really “just” a reference to know what one should be building, respectively what one should be building on.

I wrote this document in a text-based mark-up langage, in this case AsciiDoc, because that’s what we were using elsewhere too, and put it into a git repository so that it can be treated like code. Because this document isn’t set in stone, and evolves as new features get added, existing features get modified etc.

By having this dedicated repository, there is a place to draft and discuss changes to the specification.
(show this)
And because the specification carefully discusses all that is relevant to users of the system, and nothing more, this repository now crates just the right forum where the engineers *building* the system, and the engineers building *on* the system, can put their heads to gether and nail down how features should look like. This balance is very useful, because there is always a tendency to push complexity “over the border”. So if the engineers working on the internals get to define the interface on their own, it may have a tendency to be too low-level, too complicated, not general or abstract enough. And designing the feature without having the current implementation in mind helps you to think ahead, and at last try to shape the interface in a way that may stay while the implementation evolves.

As you can imagine, it took a while to anchor this document and the process around it in the organization, and it is a continuous struggle to keep everyone on board, even if it slows things down. Because even if there is that document that should be maintained, when there is pressure from above to “get things done”, it happens easily that the feature is implemented first, and then someone deigns to update the spec later. At that point, it’s becomes merely documention.

Reference implementation

So we see that paper is patient, but not pedantic. So a specification document, whether prose or more formal, can easily be ignored -- by accident or willfully, out of laziness or of spite. To combat this, it can pay off to put a reference implementation next to the specification:

So take the specification, and build the most simple, most elegant, most direct implementation that fulfills the specification. Ignore all the concerns that make your production implementation complex, and where your secret sauce may be hidden: Ignore performance, reliability, scalability, persistence, crash-recovery, distribution, security. But still fulfill the specification, and provide an compatible interface.

Now you suddenly have *two* implementatios of your specification. Both are "real enough" that they can be used by the same clients (at least for small, simple tasks and tests). So everytime something works with one implementation, but not the other, or behaves differently, you know that there is a bug in one of the implementations, and you can consult the specification which one is right. This helps to keep the production implementation honest, and keeps it from deviating from the spec.

When writing the reference implementation, I suggest you should resist sharing code with the production implementation, because a bug in the shared code would now go un-noticed! A good way to do that is to even use a different language, which makes sense anyways, given that the goals here are quite different.

Personally, I can of course highly recommend Haskell for this task, and that’s what I was using. We already had the spec for half a year, but were facing the issues with spec-just-on-paper mentioned above. So at some point I sat down and implemented the Spec in Haskell. Haskell was great because I could very rapidly build this thing, use good libraries to make it actually work – in my case, HTTP, CBOR, Crypto and WebAssembly libraries, and because it is a very high level language, I managed to mostly separate plumbing code that deals with low-level technicalities from the spec-relevant code, which should follow the specification as closely as possible.

Now those teams who have to build *on* the platform have two targets to develop and test against, and it gave them increased confidence in demanding that the production code *really* behaves like agreed on in the spec. Also, because the Haskell code is smaller and simpler, it could also be used as a library to build other more, e.g. a sandbox to test services in a programmatic way.

Conformity tests

Now that you started to write code around the spec, you can go a step further, and write a "specification acceptance test suite": This is again code that is written with *just* the specification document in mind, but sits on the "other" side and *uses* your system, according to the interface described in the specification, and probes its behavior in all the different ways described by the spec.

By using an expressive language with good testing libraries (did someone say Haskell) you may even be able to do extensive fuzzing, property-based testing with random inputs, etc.

You can use this test suite then to help development of both reference and production implementation, and it even more ensures that the specification is adhered to: A red cross in the CI report is very effective in nudging developers to keep the specification (at least the part represented in the test suite) honest.

In my case, the test suite ended up quite substantial, with more than 600 individual tests. And indeed, it was great to use Haskell here, because it meant that one or two lines could express quite complicated test. I’m particularly proud of an idea that I call the “universal service” – instead of writing separate WebAssembly modules to upload for each test, which would hardly scale, I wrote a “universal” one that implements a very simple scripting language, and this way I can code, in my Haskell tests, not just how the test program itself interacts with the platform, but seamlessly also what the installed service should do.

Bonus material

This is roughly as far as I managed to push this development model in practice. If you want to go further, here are some of the extensions of this idea that I wished I could do:

At some point I hoped that the reference implementation's code would be so elegant, so readable, so abstract that it could *be* the specification, and be understood even by readers not familiar with the language at hand. I believe this is achievable with enough effort, in particular in languages that allow you to massage the syntax a lot (Agda, Isabelle with LaTeX export). In my concrete case, using Haskell, I got close, but in the end not close enough, and maybe the biggest hurdle was the lack of elegant record syntax.

A clearly much more ambitious goal would be to take the artifacts describe above (i.e. the formal model in the specification document, the reference implementation, the conformance test suite and the production implementation) and connect them using formal methods. This is a spectrum, spanning from an isolated formal model of the abstract system that allows you to prove properties of it, to a fully verified implementation. I even did, at one point, used a tool to turn the Haskell code into Coq code so that I could load it into Coq, but didn’t get far, and the code has grown a lot since. So far, the practicalities of this were unfavorful, but maybe eventually...

If I could tick off this box, then the specification would becomes “maximally useful” in the terms described by the DeepSpec project, which requires specifications to be rich (so with detail and describing interactions), two-sided (so connected to both implementation and clients), formal (written with proper mathematical notation) and live (executable and connected by proofs to the implementation). Maybe a decade or two that level of rigor will be sustainable for a small start-up.

Related work

As I mentioned, little of what I said here is new. Some keywords to look for if you want to read about more ways to design your programs around a specification that is useful to the customer, you can look for “Consumer-driven contract testing”, “Specification by Example”, “Domain-driven Design” or the idea of “Functional Essence”.

Closing remarks

So what can you take home from this talk? Maybe the most important take-way is that as you design your system, also picture it from the outside, and create a mental model that is independent of the implementation, more abstract. And then build your system to fill that picture, rather than the other way around.

With that I thank you for your attention and am looking forward to a live and lively QA session now.