Antipattern: Misapplied Genericity

Author(s): Sven

Building a solution that is so generic that it ends up satisfying no one.

Description

We have a very specific problem to solve, and we don't only want to solve this very specific problem. That would be too easy. We want to solve the whole problem class our problem belongs to. Then our life will be easier in the future. All future problems of that problem class are solved by our generic solution to the first problem. We only need to write a little extension point here or add some configuration there and voilà, our future problem will be solved. Unfortunately it doesn't work that way, and we will end up with a very bad solution for our specific problem and a very bad solution of our future problems. When we look how researches find generic solutions to a problem class, then we see that they always find a specific solution which only works for one specific problem. If that works, they try the solution for another problem of the same problem class. If that works, usually after a few adaptions, they see a clearer picture of solution space. But still, they find another problem of that problem class and try to solve it until they know that they have found a generic solution. In software, we often hear the term "use before reuse", which works similar. First solve your very specific problem and if that works you may think about a solution which might be reusable. If we start too early with a generic solution of a problem we do not fully understand yet, we cannot find a good solution.

What are some examples?

- The Financial Company App Productline for 100+ countries
- Super-generic framework in logistics
- Global ecommerce B2B offering
- Generic Product model for 12 insurance products
- App Development with a cross platform framework

Why does this happen?

Stakeholders get the same idea: "we could build this functionality in a generic way so that we can easily reuse it elsewhere in the future". Solve the problem once and reuse the solution everywhere. Some stakeholders, like project managers, see the chance of saving future cost. Other stakeholders, like developers, like to solve very hard problems. To start the downward spiral of generic development you need only one stakeholder asking for a generic solution. It is very hard to convince the stakeholders, who are in favour of the idea, to throw it overboard, because the benefit is easily overrated and the cost is easily underrated.

How can we avoid getting into the situation in the first place?

When developers get confronted from a stakeholder to implement such a generic solution, we need to explain why a generic solution never works from the beginning and also relate to how scientists find generic solutions. Developers also need to be aware that the those stakeholders are disappointed and may look for someone else who just says “yes, we can do that for you”. Therefore, developers need to be really convincing. On the contrary, developers should always solve the specific problem first. Whenever a similar problem of the same problem class comes up again, developers need to think carefully if they should work on a generic solution. Sometimes it is never vital to implement a generic solution, because it either does not exist or is too expensive to implement and maintain.

What are suggestions to get out of the situation if we ended up in it?

When a solution is too generic, it is vital to make it more specific. This can either be done by developing a completely new and very specific solution from scratch (a rewrite) or by finding an intermediate solution. A complete rewrite is most of the time a terrible idea, see Things You Should Never Do or The Big Rewrite. We need to be careful with a rewrite, because the effort of it is almost always totally underestimated. It becomes even more expensive if you have many customers of your generic solution, and you need many specific rewrites for each of your customers. Another problem with the rewrite is that your stakeholders possibly won’t trust you anymore. You already spent a lot of money on a solution which is not working and now you want again a lot of time and money to fix the problems you caused in the first place. Stakeholders usually want to see results rather quickly, therefore it is vital to understand your biggest problem and fix that first. Then the 2nd biggest problem, and so on. Build a prototype to validate your improvement idea and then scale it. Good software engineering practice is good for you: improve and deliver incrementally and iteratively. Working in increments without thinking deeply about your architecture may lead to the fact that you pencil yourself in a corner where you don’t get out easily. Therefore only start your improvements if you have a good picture of the overall solution. In other words: do classic architecture work.