Yagni

26 May 2015

Martin Fowler

PROCESS THEORY
PROJECT PLANNING
EVOLUTIONARY DESIGN
PROGRAMMING STYLE

Yagni originally is an acronym that stands for "You Aren't Gonna Need It". It is a mantra from ExtremeProgramming that's often used generally in agile software teams. It's a statement that some capability we presume our software needs in the future should not be built now because "you aren't gonna need it".

Yagni is a way to refer to the XP practice of Simple Design (from the first edition of The White Book, the second edition refers to the related notion of "incremental design"). [1] Like many elements of XP, it's a sharp contrast to elements of the widely held principles of software engineering in the late 90s. At that time there was a big push for careful up-front planning of software development.

Let's imagine I'm working with a startup in Minas Tirith selling insurance for the shipping business. Their software system is broken into two main components: one for pricing, and one for sales. The dependencies are such that they can't usefully build sales software until the relevant pricing software is completed.

At the moment, the team is working on updating the pricing component to add support for risks from storms. They know that in six months time, they will need to also support pricing for piracy risks. Since they are currently working on the

pricing engine they consider building the presumptive feature [2] for piracy pricing now, since that way the pricing service will be complete before they start working on the sales software.

Yagni argues against this, it says that since you won't need piracy pricing for six months you shouldn't build it until it's necessary. So if you think it will take two months to build this software, then you shouldn't start for another four months (neglecting any buffer time for schedule risk and updating the sales component).

The first argument for yagni is that while we may now think we need this presumptive feature, it's likely that we will be wrong. After all the context of agile methods is an acceptance that we welcome changing requirements. A plan-driven requirements guru might counter argue that this is because we didn't do a good-enough job of our requirements analysis, we should have put more time and effort into it. I counter that by pointing out how difficult and costly it is to figure out your needs in advance, but even if you can, you can still be blind-sided when the Gondor Navy wipes out the pirates, thus undermining the entire business model.

In this case, there's an obvious cost of the presumptive feature - the **cost of build**: all the effort spent on analyzing, programming, and testing this now useless feature.

But let's consider that we were completely correct with our understanding of our needs, and the Gondor Navy didn't wipe out the pirates. Even in this happy case, building the presumptive feature incurs two serious costs. The first cost is the cost of delayed value. By expending our effort on the piracy pricing software we didn't build some other feature. If we'd instead put our energy into building the sales software for weather risks, we could have put a full storm risks feature into production and be generating revenue two months earlier. This **cost of delay** due to the presumptive feature is two months revenue from storm insurance.

The common reason why people build presumptive features is because they think it will be cheaper to build it now rather than build it later. But that cost

comparison has to be made at least against the cost of delay, preferably factoring in the probability that you're building an unnecessary feature, for which your odds are at least ⅔. [3]

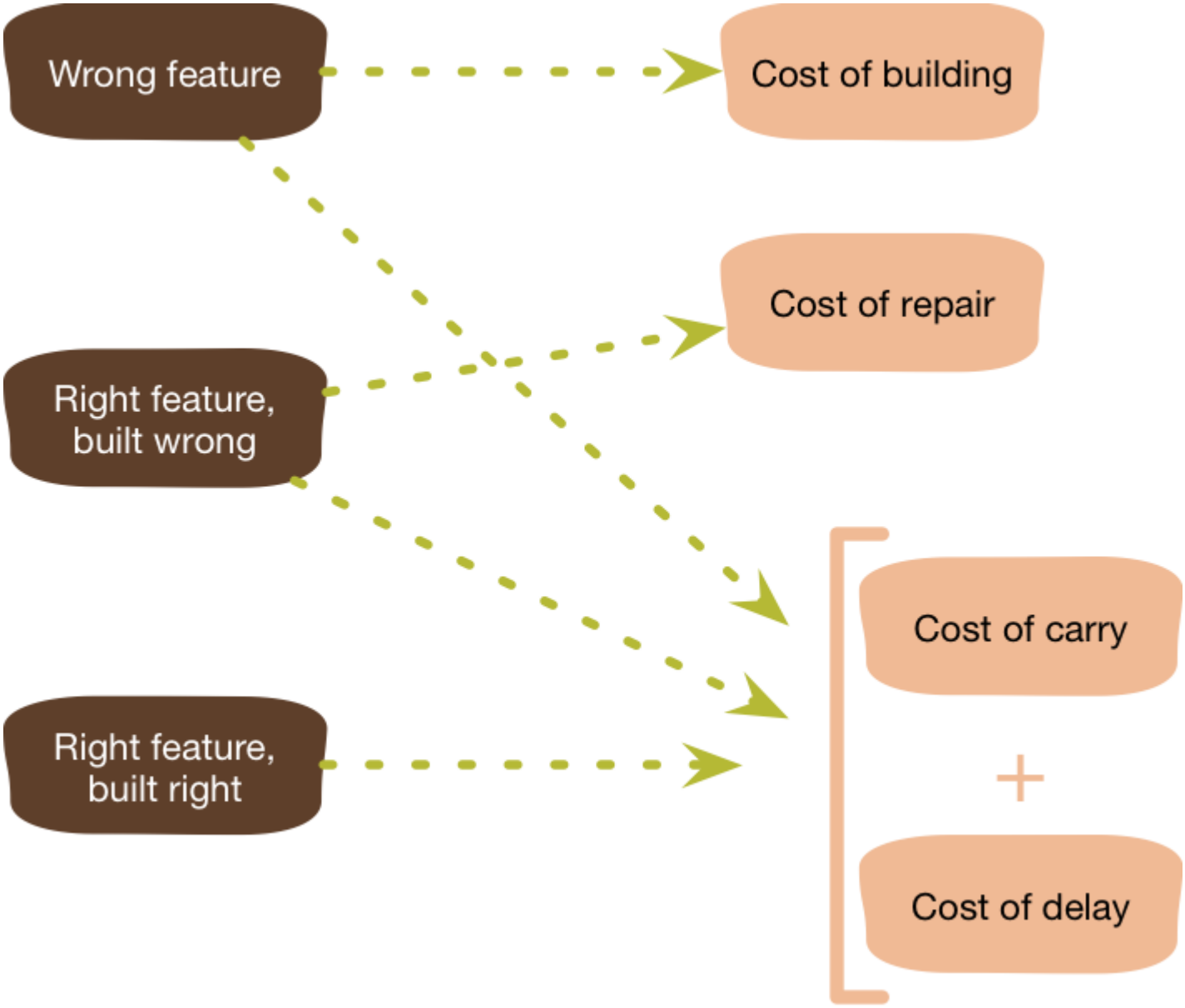
Often people don't think through the comparative cost of building now to building later. One approach I use when mentoring developers in this situation is to ask them to **imagine the refactoring** they would have to do later to introduce the capability when it's needed. Often that thought experiment is enough to convince them that it won't be significantly more expensive to add it later. Another result from such an imagining is to add something that's easy to do now, adds minimal complexity, yet significantly reduces the later cost. Using lookup tables for error messages rather than inline literals are an example that are simple yet make later translations easier to support.

The cost of delay is one cost that a successful presumptive feature imposes, but another is the **cost of carry**. The code for the presumptive feature adds some complexity to the software, this complexity makes it harder to modify and debug that software, thus increasing the cost of other features. The extra complexity from having the piracy-pricing feature in the software might add a couple of weeks to how long it takes to build the storm insurance sales component. That two weeks hits two ways: the additional cost to build the feature, plus the additional cost of delay since it look longer to put it into production. We'll incur a cost of carry on every feature built between now and the time the piracy insurance software starts being useful. Should we never need the piracy-pricing

software, we'll incur a cost of carry on every feature built until we remove the piracy-pricing feature (assuming we do), together with the cost of removing it.

So far I've divided presumptive features in two categories: successful and unsuccessful. Naturally there's really a spectrum there, and with one point on that spectrum that's worth highlighting: the right feature built wrong. Development teams are always learning, both about their users and about their code base. They learn about the tools they're using and these tools go through regular upgrades. They also learn about how their code works together. All this means that you often realize that a feature coded six months ago wasn't done the way you now realize it should be done. In that case you have accumulated TechnicalDebt and have to consider the **cost of repair** for that feature or the on-going costs of working around its difficulties.

So we end up with three classes of presumptive features, and four kinds of costs that occur when you neglect yagni for them.



My insurance example talks about relatively user-visible functionality, but the same argument applies for abstractions to support future flexibility. When building the storm risk calculator, you may consider putting in abstractions and parameterizations now to support piracy and other risks later. Yagni says not to do this, because you may not need the other pricing functions, or if you do your current ideas of what abstractions you'll need will not match what you learn when you do actually need them. This doesn't mean to forego all abstractions, but it does mean any abstraction that makes it harder to understand the code for current requirements is presumed guilty.

Yagni is at its most visible with larger features, but you see it more frequently with small things. Recently I wrote some code that allows me to highlight part of a line of code. For this, I allow the highlighted code to be specified using a regular expression. One problem I see with this is that since the whole regular expression is highlighted, I'm unable to deal with the case where I need the regex to match a larger section than what I'd like to highlight. I expect I can solve that by using a group within the regex and letting my code only highlight the group if a group is present. But I haven't needed to use a regex that matches more than what I'm highlighting yet, so I haven't extended my highlighting code to handle this case - and won't until I actually need it. For similar reasons I don't add fields or methods until I'm actually ready to use them.

Small yagni decisions like this fly under the radar of project planning. As a developer it's easy to spend an hour adding an abstraction that we're sure will soon be needed. Yet all the arguments above still apply, and a lot of small yagni decisions add up to significant reductions in complexity to a code base, while speeding up delivery of features that are needed more urgently.

Now we understand why yagni is important we can dig into a common confusion about yagni. **Yagni only applies to capabilities built into the software to support a presumptive feature, it does not apply to effort to make the software easier to modify.** Yagni is only a viable strategy if the code is easy to change, so expending effort on refactoring isn't a violation of yagni because refactoring makes the code more malleable. Similar reasoning applies for practices like SelfTestingCode and ContinuousDelivery. These are enabling practices for evolutionary design, without them yagni turns from a beneficial practice into a curse. But if you do have a malleable code base, then yagni reinforces that flexibility. Yagni has the curious property that it is both enabled by and enables evolutionary design.

> *Yagni is not a justification for*
> *neglecting the health of your code base.*

Yagni requires (and enables) malleable code.

I also argue that yagni only applies when you introduce extra complexity now that you won't take advantage of until later. If you do something for a future need that doesn't actually increase the complexity of the software, then there's no reason to invoke yagni.

Having said all this, there are times when applying yagni does cause a problem, and you are faced with an expensive change when an earlier change would have been much cheaper. The tricky thing here is that these cases are hard to spot in advance, and much easier to remember than the cases where yagni saved effort [4]. My sense is that yagni-failures are relatively rare and their costs are easily outweighed by when yagni succeeds.

Further Reading

My essay Is Design Dead talks in more detail about the role of design and architecture in agile projects, and thus role yagni plays as an enabling practice.

This principle was first discussed and fleshed out on Ward's Wiki.

Notes

1: The origin of the phrase is an early conversation between Kent Beck and Chet Hendrickson on the C3 project. Chet came up to Kent with a series of capabilities that the system would soon need, to each one Kent replied "you aren't going to need it". Chet's a fast learner, and quickly became renowned for his ability to spot opportunities to apply yagni. Although "yagni" began life as an acronym, I feel it's now entered our lexicon as a regular word, and thus forego the capital letters.

2: In this post I use "presumptive feature" to refer to any code that supports a feature that isn't yet being made available for use.

3: The ⅔ number is suggested by Kohavi et al, who analyzed the value of features built and deployed on products at microsoft and found that, even with careful up-front analysis, only ⅓ of them improved the metrics they were designed to improve.

4: This is a consequence of availability bias

Acknowledgements

Rachel Laycock talked through this post with me and played a critical role in its final organization. Chet Hendrickson and Steven Lowe reminded me to discuss small-scale yagni decisions. Rebecca Parsons, Alvaro Cavalcanti, Mark Taylor, Aman King, Rouan Wilsenach, Peter Gillard-Moss, Kief Morris, Ian Cartwright, James Lewis, Kornelis Sietsma, and Brian Mason participated in an insightful discussion about drafts of this article on our internal mailing list.