Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pragmaticprogrammer.com ■ dave@pragmaticprogrammer.com

The Trip-Packing Dilemma

Andy Hunt and Dave Thomas

A guiding principle behind agile software development is the idea of keeping things simple. The Agile Manifesto defines it this way: "Simplicity—the art of maximizing the amount of work not done—is essential."

We want to actively increase the "amount of work not done." We do this by only implementing essential features or framework functionality

and by taking steps (such as constant refactoring and avoiding "broken windows"[1]) to avoid future problems that might require extra work.

Making life simpler is an admirable goal, but unfortunately many developers have a knack for making one of two errors:

- Oversimplifying something that really is complex
- Overcomplicating something that should be easy

The first problem is usually readily noticeable as long as a customer or sponsor is involved in the project, as the corresponding lack of functionality soon becomes apparent. Even if the problem isn't noticeable at the user level, you might see warning signs in the code: the number of "special cases" needed to handle a particular piece of application functionality starts to grow suddenly. You start to see code that just isn't tractable; pushing on it "here" causes another problem to pop up over "there." Developers might begin to exhibit that sudden stricken look (like a deer caught in the headlights of a rapidly approaching car) as they proceed further into the requirements, and so on. We'll talk more about this "Whack-a-Mole" style of development in a later column; for now, onto the second problem.

Extreme programming attacks overembellishment with the YAGNI (you aren't gonna need it) principle.[2] This pithy maxim admonishes us to build today only those features that we truly need today. The notion that "we might need this" later turns out to be false so often that's it better to err on the conservative side and never build anything unless you absolutely, positively need it right now.

But is that notion true? Does it really work that way?

Some have criticized XP on this issue, saying that YAGNI limits intelligent foresight. Taken at face value, it might seem that way, but as Kent Beck has remarked, "YAGNI is not a license to be stupid." So perhaps we need to look at this issue more pragmatically: how do we decide whether to build something that isn't strictly necessary today but could be cheaper in the long run to build (or start) now and could save us from larger pains later on?

Deciding what to pack

One way to look at this issue is to consider how you would pack a suitcase or overnight bag for a short trip. You wouldn't want to head into the jungle or other hostile area unprepared, but then again you have to travel light. The carrying cost of small items in a bag (the extra weight and volume used) is pretty much negligible until it exceeds some threshold. All of sudden, the bag seems too heavy to lift comfortably. Now you've got some decisions to make: What do you absolutely need? What *might* you need? What's the cost of taking it and of not taking it?

Experienced backpackers might well laugh at the novice who brings along the proverbial kitchen sink. But at the same time, omitting even something as small as a first aid kit might prove fatal. As with most decision-making processes, it's an exercise in risk management. Here's where it gets interesting: unlike the overnight bag, repacking code is itself another expense. And the costs of even a little bit of extra weight are far more significant. The carrying cost of something as minor as an extra unused method in a class is often much more than we realize. All code—even unused code—must be written, understood, documented to some degree, tested, and maintained as the system evolves. Systemic overbuilding can quickly lead to the software equivalent of a bag you can't lift.

Overbuilding can also indicate deeper misunderstandings. One fellow on a previous client's team (we'll call him Dick) took great care—and copious amounts of code—to build a class to handle an interval of time. He included appropriate conversion operators to convert this integer value to every imaginable basic data type: short, long, double, float, maybe even a char (don't ask).

None of this was ever used because the business definition of the quantity meant it had to be treated as a native integer anyway. Worse yet, because of the many interesting business rules that governed the handling of this quantity, it should never have been exposed as a native type in the first place.

But Dick's approach is a perfect example of how YAGNI should apply: he was taught the canonical dogma that an abstract data type should provide operators for type conversion to all possible native data types—just in case you might need them. That sort of wasteful practice leads to geometric code bloat and correspondingly increased costs.

The other side of the coin is just as dangerous, even though it doesn't have a catchy slogan associated with it yet. Toward that end, we humbly suggest the new acronym DOGBITE: *do* it, *or g*et *b*itten *i*n *t*he *e*nd. Here's how it works.

Dick's coworker Jane read about XP once and is a strong advocate of YAGNI. Jane is working on support code to save and restore the state of serialized objects. For the moment, no requirement exists to support transactions, although that will be coming. It's hard to envision how that would work for now, so Jane puts off thinking about it. After all, it's not needed right now.

Several months later, the time comes when a user-visible feature now requires transaction support. Unfortunately, because the earlier design did not allow for transactions, what might have been a simple matter of adding functionality becomes a large, messy endeavor that finally costs the team a complete rewrite of this part of the code base. A small arithmetic cost incurred earlier could have spared us a larger geometric cost now.

Social aspects

Beyond the pure technical issues, you can also have reasons to invest beyond today's requirements brought on by "softer" considerations. Suppose that prior to implementing a feature, you know that the customer has changed her mind—drastically—several times among several possibilities. Does it sound prudent to go ahead and code the feature according to the last way the wind blew? Or, might this be an occasion to invest in a more flexible approach, using metadata, for instance? There's no strict requirement for metadata, but your intuition tells you that this area is volatile and could change yet again.

As with any investment, it's a gamble you could lose. The last decision could remain the last decision, and you'd lose the extra effort invested. Then again, the added flexibility might save you several days or even weeks of rework down the line if you're right.

But there's a funny nonlinearity about this decision. Looking back at the last several projects we've dealt with, a consistent trend seems to emerge. The portions of systems that went beyond the requirements of the day—those that heavily used metadata, configurable options, and the like—enjoyed a longer and more robust life than those that didn't. In a recent case, the only portion of a large system that survived past the first year was the one we built. It was driven by its own custom, domain-specific language, which was implemented using *yacc* and *lex*. Inherently flexible beyond the immediate requirements, it alone survived.

These techniques don't suit every project, of course. The only way to find out whether they do or don't is to use your own judgment, regardless of any current pithy maxims.

Limitations of pith

The pragmatic approach always takes advantage of experience. Your experience, after all, is where your value lies. It's how you keep your job, or get the next one. Taking a pithy maxim such as YAGNI to extremes dehumanizes the development process and cuts off your foresight born of experience.

Pithy maxims just remind us of deeper issues; a world of gray matter exists behind every black-and-white rule. That's why there's no substitute for experience, judgment, and common sense. Without these, and despite the best practices and latest methodologies, project teams will fail.

Don't shortchange yourself or your team: listen to your intuition, learn from your experience. That's the pragmatic thing to do.

References

1. A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 2000.
2. K. Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.

Andy Hunt and Dave Thomas are partners in The Pragmatic Programmers. They feel that software consultants who can't program shouldn't be consulting, so they keep current by developing complex software systems for their clients. Contact them via www.pragmaticprogrammer.com.