Cutting with the Grain: The Rhythms of Design

Kent Beck[1]
Joseph Leddy
William Wake

One way to make software development more valuable is to spend as much time as possible going "with the grain". Programming languages, personal style, and team dynamics all create some paths in which development flows smoothly. They also create other paths which are bumpy and potholed. This paper describes how you can identify which kind of path you are on and what you can do when you find yourself on a bumpy path.

Sometimes programming seems challenging but it flows. Every step forward makes the next step forward easier. Success builds on success. Sometimes programming is hard and frustrating. Every step forward is like wading in wet concrete, and the concrete is setting. The last change you made interferes with this change and this with the next. Identifying which rhythm of development you are working in is important because the strategies for optimal design are different, sometimes opposite, in the two cases.

In a recent Programming Intensive[2] at the Three Rivers Institute we were playing with code for determining if a game of Tic-Tac-Toe is finished. The code started out looking like this:

```
public boolean over() {
    for (int x = 0; x < board.length; x++)
        if (board[x][0] != ' ' && board[x][0] == board[x][1]
            && board[x][0] == board[x][2]) return true;
    for (int y = 0; y < board[0].length; y++)
        if (board[0][y] != ' ' && board[0][y] == board[1][y]
            && board[0][y] == board[2][y]) return true;
    if (board[0][0] != ' ' && board[0][0] == board[1][1]
        && board[0][0] == board[2][2]) return true;
    if (board[0][2] != ' ' && board[0][2] == board[1][1]
        && board[0][2] == board[2][0]) return true;
    return false;
}
```

It was ugly. There was a lot of duplication between the cases. We thought there had to be a different approach that would yield simpler code. We started by replacing the code that recognized a horizontal win with a more data-driven approach. Here's what we came up with after about ten minutes:

```
public boolean over() {
    int [][] offsets= {{0, 0}, {0, 1}, {0, 2}};
    for (int x= 0; x < board.length; x++)
        if (board[offsets[0][0]][offsets[0][1]] != ' '
            && board[offsets[0][0]][offsets[0][1]] ==
                board[offsets[1][0]][offsets[1][1]]
```

[1] The authors can be reached at *mailto:kent@threeriversinstitute.org*, *mailto:josephleddy@juno.com*, *and* *mailto:william.wake@acm.org*, *respectively.*

[2] www.threeriversinstitute.org

```
            && board[offsets[0][0]][offsets[0][1]] ==
                board[offsets[2][0]][offsets[2][1]])
                return true;
    for (int y= 0; y < board[0].length; y++)
        if (board[0][y] != ' ' && board[0][y] == board[1][y]
            && board[0][y] == board[2][y])
                return true;
        if (board[0][0] != ' ' && board[0][0] == board[1][1]
            && board[0][0] == board[2][2])
                return true;
        if (board[0][2] != ' ' && board[0][2] == board[1][1]
            && board[0][2] == board[2][0])
                return true;
    return false;
}
```

This code looked worse than the problem code we were trying to replace. It was harder to read and harder to understand. It wasn't obvious how to generalize it to cover the other cases. At this point we stopped and went back to the original code. Going further, pushing harder, would have been counterproductive. Neither our programming language (Java), nor our chosen representation of the board, nor our collective experience offered a way to make the code any cleaner. If we were working in a different language or had represented our board differently or knew "the trick", our proposed transformation might have resulted in clean code. As it was, more effort seemed like it would just result in more mess.

There was a pattern here, one that repeats at many different scales of development. Sometimes, trying to add functionality or generalize a program makes it better—clearer, faster, smaller, more flexible. Sometimes, trying to generalize makes a program worse. Identifying these rhythms and responding appropriately makes programmers more effective. Trying to push the snowball uphill is a losing battle—the farther you push, the bigger the ball becomes and the greater the resistance. If you can find a way to push the snowball downhill, though, each step makes the next step easier.

These two rhythms of work, which we call "with-the-grain" and "across-the-grain", are characteristic of both problems and their solution. Failing to recognize which rhythm of work one is in results in wasted effort and wasted opportunity.

You can see the cross-grain effect when working with an old system without automated tests. Every defect creates two more. Developing with the grain, you seem to find defects before they affect customers. Every refactoring simplifies the code and leads to further simplification.

Part of recognizing which rhythm you are in is an intellectual and objective awareness. We could see in our Tic-Tac-Toe example that the code was getting longer, not shorter. Developing with the grain, code is typically shrinking or at worst growing slowly compared to the growth in functionality. Developing across the grain, the code grows faster than the functionality. One has the sense of climbing a sand dune—lots of motion but little progress.

You also get emotional feedback about how your work is going. Developing with the grain is a joy. Background worries about aspects of the code disappear as complexity vanishes. A growing sense of relief accompanies the maturation of the design. The emotions are quite different when you're developing across the grain. You develop a sense of hopelessness as the code deteriorates. Frustrations grow—this just *shouldn't* be this hard.

When you are working with the grain, exploit and enjoy it. As long as you don't have a pressing need to move on to the next feature, keep simplifying. Investment now will likely pay off when you go to do the next thing. Eventually you will finish mining this vein of simplification and be ready to move on. Avoid the temptation to generalize beyond your actual needs. Speculative abstraction most often has to be ripped out or worked around when reality proves different than what you imagined it would be.

When you find yourself working across the grain, the strategy is more complex. Once you are aware of the situation, you have options to cope with cross-grain development:

- Abandon
- Amputate
- Push on
- Transform
- Wait

When you are working across the grain, one strategy is to *abandon*: stop trying to add the feature at all. This transforms the design/implementation problem into a requirements/analysis problem. Do you really need the feature? What are the consequences of leaving it out? You will likely have to communicate with your customer if you intend to abandon a feature.

If you *do* abandon a feature, methodically remove all traces of it from the system. Finding the vestiges of a no-longer-used feature causes tension. Tearing out a vestigial feature later in development has unpredictable effects. Leaving it in slows development. If, instead, you are careful to clean up abandoned features, the system stays as clean as it can be for the next feature. Cleanup is part of the cost of abandonment.

A more daring strategy for dealing with cross-grain development is *amputation*[3]. In abandonment you stop adding functionality in a certain direction. In amputation you abandon and remove all similar features. If a little is bad, perhaps none at all would be best.

An example of amputation is the security policy in Ward's Wiki. The collaborative nature of a Wiki makes it tempting to add security: editors of a page should at least have to log in first, shouldn't they? Once you start adding security features, though, it's hard to see where to stop. Instead, Ward chose to eliminate all technical notions of security from the Wiki.

Successful amputation is rare and precious. Amputation is rare because it requires a new way of looking at a problem and the courage to leave out conventionally-demanded features. Amputation is precious because the effort not expended on a class of difficult-to-implement features can be turned to create innovative features along some other dimension.

A large-scale application of amputation is what *The Innovator's Dilemma*[4] calls "changing the basis of competition". Small entrants to a marketplace necessarily have to pick their battles. One way to do this is to amputate the strengths of competitors from your product, staying with the grain for the features you do implement, and building a compelling product that won't or can't be compared directly to established competitors. JUnit did this by using Java for its scripting language instead of the proprietary scripting

[3] We took "amputation" from Edward deBono's book *Simplicity* (Harper-Collins, 1999, ISBN: 0140258396). DeBono talks about "provocative amputation" as one way of simplifying a complex situation.
[4] Clayton M. Christensen, *The Innovator's Dilemma*, HarperBusiness, 2003, ISBN: 0060521996.

languages of other testing tools. JUnit used its reduced complexity to lower the cognitive footprint of testing so programmers could imagine easily writing tests.

Another strategy in response to cross-grain development is to *push on*. You might think you have almost found the idea that will make a feature easy to add. Your customer might demand a feature. You might think you have no choice. If you have to implement a difficult feature, clear your mind and get to work. Sometimes work is work, hard work. Fussing doesn't help. Don't make more of a mess than you need to. Remember this as a spot to consider for future simplification. Don't invest more than necessary at this time. Simply pushing on without thought is seldom a good idea: progress usually requires a change in attitude, perspective, or approach.

Another strategy for dealing with cross-grain development is to *transform* the situation through the introduction of new ideas, technology, or people. Part of the magic of programming is that just the right idea can turn yesterday's hard slog into tomorrow's luge run to success. For example, suppose you are working on a scripting language. It is getting harder to extend because the parser is getting messy. The same code can become a joy to work with again through the introduction of a parser generator. Our Tic-Tac-Toe scoring algorithm might be a breeze if we stored the board differently or if we reimplemented it in a language with strong pattern matching features like Perl.

Transforming cross-grain development by introducing new technology is not a purely technical decision. In a team already familiar with parser generators, introducing one is likely a good idea. If no one on the team knows parser generators, though, then you need to decide whether the cost of learning the technology is worth the benefits. Creating a ghetto in the system that only one team member understands doesn't help you go with the grain for long.

Another form of transformation is transforming yourself. *The Pragmatic Programmer*[5], for example, has many ideas for expanding your personal "sweet spot". The more techniques you have at your fingertips, the more problems you can solve smoothly.

Finally, *waiting* is sometimes the best strategy for incubating a transforming idea. Get a coffee, take a walk, go home, talk to a friend, do something else for a few days; these are activities that can give your mind the space it needs to create and bring to consciousness the idea that makes development smooth again. Physical activities seem particularly effective at encouraging ideas.

It won't always be obvious before you start programming a feature whether your development will go with the grain or across the grain. Until the feature is satisfactorily completed, remember that you have options. Stubborn commitment to a single path can blind you to a better solution. Maintain awareness of the process and how it is going. If you can safely abandon features that don't work out, you can make development more valuable. However, making the scope of the system fluid requires communication and trust among the whole team.

You can improve the overall effectiveness of development by working "with the grain" of your programming language, your technology, your team, and yourself. When the development of a feature begins to go badly, think about whether you have to solve this problem now or solve this problem at all or if a different approach to solving it would

[5] Andy Hunt and Dave Thomas, *The Pragmatic Programmer*, Addison-Wesley Professional, 1999, ISBN: 020161622X.

go more smoothly. If you decide to proceed in spite of the difficulty; take a break, consider your options, take the time you need to make good decisions, get the information you need, and then proceed with joy.

Acknowledgements

Thanks to Andrew Hunt, Steve McConnell, and Laurent Bossavit for their comments on a draft of this paper. Thanks to Cindee Andres for masterful editing. Any remaining errors or stupidities are the sole responsibility of the authors.