A Means to an End — Getting the Job Done

Gertjan Zwartjes
gertjan.zwartjes@gmail.com

Abstract. Failing software projects have plagued software development ever since its existence. Numerous different treatments for this plague have been introduced, where the latest trend is product centred and focuses on customer satisfaction. Whereas none of the methods, tools and techniques are a Silver Bullet, many are important aids to a successful software project. However, there is an important pitfall concerning all software engineering tools, methods and techniques: when a means is promoted to become an end. To prevent this from happening common sense and alertness are the key.

Index Terms. Programming, Software Engineering, Software Development Management

I. Introduction

THE most lively trend in current software engineering seems to be the paradigm of "getting the job done." Inspiring books like The Pragmatic Programmer" [1] or "Facts and Fallacies of Software Engineering" [2] and popular bloggers like Joel Spolsky [3] or Jeff Atwood [4] focus on the practical aspects of surviving software projects. Also, recent software processes, such as the ones promoted by the Agile movement, refocused on customer satisfaction, instead of being more process oriented like the "older" methods. What makes those books hard to put down, and bloggers like Spolsky and Atwood so popular? Is there a more general idea behind this trend? Maybe there is.

Let me start with a question: "Who is paying the salaries of those of us who are working on a software product?" It's not our boss. Not the company we work for. It's that same customer that Spolsky, Atwood and the Agile folks are talking about. Just like the sun is the source of all energy on planet earth, the customers are the main money source of us who produce software. So they'd better be happy, else we might run out of business really fast. Sometimes we slowly seem to forget this, although deep down, we all know that a product that meets his needs is what makes a customer smile. Just remember that this is what makes him pay for our work. In fact, to put it very bluntly, this is why "software engineering" was invented in the first place; software development was—and unfortunately in many cases still is—chaotic: projects being late, over budget and deficient in functionality.

The field of software engineering has flourished in the past decades. It started in the 1960s, and was much stimulated by the software crisis of the 60's, 70's and 80's, when many software projects had bad endings. Strict engineering processes were introduced and a whole new field of research emerged. Most notably, in 1984 the Software Engineering Institute (SEI) was established, which has been the origin of many ground breaking initiatives in software engineering. Besides methodology, we have seen major advances in programming languages too. For example the movement from sequential languages such as C and Pascal to object oriented languages such as C++ or Java. According to Fred Brooks himself: "I think object-oriented programming in fact lifts design thinking to a higher level and thus addresses the essence of the problem" [7]. In addition, we have also seen incredible improvements in tool support for the process of software development and software development itself. Among others, tools for version control, automated building, automated testing and the like have become indispensable in the modern software developer's tool-belt.

All these improvements in software engineering unfortunately have a downside too: *the worst thing that can happen is that a means is being promoted to an end.* I will try to explain this with examples considering four different topics: processes, tools, testing and coding. Then I will shortly discuss ways to avoid this pitfall and the presence of the same problem in software engineering research. I will conclude with an analogy between running a restaurant and running a software project.

II. Means Becoming an End

Let me start with the example where a software engineering process as a means is promoted to an end. I have been in a project where the process was considered to be almost holy. Almost all work was in the service of the process: a crude violation of the first value of the Agile Manifesto [8]—"individuals and interactions over processes and tools"—commonly leading to a slipping project. Instead of working on the product to deliver, valuable time was wasted in writing and updating documents that were never read, setting up complex procedures that were not used, and all other kinds of tasks purely in service of the process.

Not only process, but tools may become holy as well. In the past, I have had to "fix" perfectly functioning and beautifully written code, to keep QA tools from choking. To keep the tools working it took me vast amounts of time after each commit. Besides that, I remember cases where I had to make the code *less* readable and *harder* to understand, which is exactly the opposite of what you would want to achieve with QA tools. I would not have complained in case these tools were vital to the project, but they were almost never used at all—which is not uncommon: "Tools are the toys of the software developer. They love to learn about new tools, try them out, even procure them. And then something funny happens. The tools seldom get used" [2].
Besides processes and tools, there is another artefact in software engineering that can be declared sacred, namely

Gertjan Zwartjes is an associate member of Espresso, author of http://www.code-muse.com, and currently employed at ASML as senior software designer. (e-mail: gertjan@code-muse.com).

the test set. And this time, I agree; I do not want to invalidate test driven development and get in a fight with the XP guys. However, there still is a pitfall here too. It is in the test benchmarks. Namely, it is the quality of the benchmark that makes or breaks a test case. It is definitely not trivial to explain why a test case fails after a source code change, but it is one thing to declare the benchmark to be holy. The danger in writing code just for the sake of making the test set run successfully, is in the quality of the test set. Therefore, it is very important to remember that the test set should be at the service of the software project and not the other way around.

Last but certainly not least: coding for the sake of coding. A common problem among software developers. As Jeff Atwood puts it so eloquently: “Software developers think their job is writing code. But it's not. Their job is to solve the customer's problem. Sure, our preferred medium for solving problems is software, and that does involve writing code. But let's keep this squarely in context: writing code is something you have to do to deliver a solution. It is not an end in and of itself” [9].

III. Stay Alert at All Times

So what is the key point here? Atwood already stated that “writing code is not an end itself” [9], however, I think this is even more general and applies to all other areas of software engineering too. All too often, I have seen that means being turned into ends.

What can you do about it? Most importantly, you should always remain aware of what you are doing and remember why you are doing it, regardless of whether you are writing code, designing, testing, writing specifications or whatever it is that you are doing for a software project. This might sound vague and yes, this is nothing different from common sense, but unfortunately, I've seen many cases where just that tiny little bit of common sense would have avoided gigantic missteps in a software project or organization. To overcome these missteps alertness is the key. Whenever you feel awkward about something you need to do in order to obey to a process or a tool, take a step back and find out why you actually need to do that in the first place: try to answer the question what the end is for the means.

I have encountered two approaches that relate to the problem I'm describing and more or less give a way to raise your alertness in this regard. The first is from “The Toyota Way” [6]. Despite the fact that car production is a very a different context, I believe the principles from the Toyota Way can be loosely mapped onto software development too. It is principle 14 that applies in the situation I'm describing: “Become a learning organization through relentless reflection (hansei) and continuous improvement (kaizen). This involves criticizing every aspect of what you do.” To this end, to determine the root cause of a problem, among others the method of the “5 Whys” is introduced in the Toyota production system [5]. But you can also use that same method to find out whether some task you are working on is helping to get the job done or not. It should be clear, after asking yourself “Why?” five times, whether the task is beneficial for the customer or not. As a result, the task may turn out to be not so beneficial, or worse, it might as well be detrimental for the success of the project. In this case, you should surely reconsider it.

Jeff Atwood writes about another approach to find out whether somebody is coding for the sake of coding, called the elevator test: “As software developers, we spend so much time mired in endless, fractal levels of detail that it's all too easy for us to fall into the trap of coding for the sake of coding. [...] every person on your team should be able to pass the elevator test with a stranger—to clearly explain what they're working on, and why anyone would care, within 60 seconds” [9].

IV. Software Nightmares

I would like conclude by introducing an analogy between running a restaurant and a software project. The last couple of weeks I've been watching the show called “Ramsay's Kitchen Nightmares”, where Gordon Ramsay helps failing restaurants to get back on the map. You either love him or hate him, but I must admit that I'm a big a fan.

In each episode of Kitchen Nightmares, Ramsay takes kind of the same approach. He enters a restaurant and starts with an investigation to see what's wrong. As you would expect, the first thing he does is ordering a few dishes from the menu to check out the quality of the food—and thus the chef. After that he will inspect the kitchen and he checks out the staff. He then tries to turn those things around that seem to be the bottlenecks for the restaurant to be successful—in his own style: unabashed to tell the truth, wearing his heart on his sleeve, and using the f-word multiple times in almost every sentence. However, if you listen carefully, you will notice that in between all the swearing, he actually knows what he is doing and that he is genuinely committed to help improve an establishment. His power lies in three things: 1) He is an outsider, he can take a step back and investigate things from a distance. 2) He wears his heart on his sleeve and he has the courage to tell the truth even if it might hurt somebody. 3) He keeps things simple—this way he can achieve higher quality and customers will be happier. On top of that, he employs his strengths to achieve but one crystal clear goal: getting clientèle and give them the best possible evening that will make them return or bring in more guests.

Back to the software world, where we have our own “Software Nightmares.” Unfortunately software development isn't sexy enough for television, however, I believe a passionate person like Gordon Ramsay, with that kind of drive, could be successful in the software world too. I know nobody will watch “Software Nightmares” for fun, but I do hope we have enough of our own Gordon Ramsay's in our field out there to turn around failing software projects.

To Derrick

Congratulations on your 60th birthday! I hope you'll remember we discussed the topic at the Espresso Workshop last year (2007), because, when Bruce and Stefan asked me to write something in honour of your birthday (which for me is a privilege to do so) I immediately recalled our talk about my presentation on the first day of the workshop. For this occasion, and with a hard deadline, I'm excited that I finally managed to put down my thoughts about our chat. For the future I wish you all the best, and I hope we will have many more such inspiring conversations.

REFERENCES

[1] Andrew Hunt and David Thomas, *The Pragmatic Programmer*. Addison Wesley, 1999.

[2] Robert L. Glass, *Facts and Fallacies of Software Engineering*. Addison Wesley, 2003.

[3] Joel Spolsky. *Joel on Software*. [Online]. Available: http://www.joelonsoftware.com.

[4] Jeff Atwood. *Coding Horror*. [Online]. Available: http://www.codinghorror.com.

[5] Taiichi Ohno, *Toyota production system: beyond large-scale production*. Productivity Press 1988.

[6] Jeffrey Liker,*The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*. McGraw-Hill, 2004.

[7] Gertjan Zwartjes (2007, Aug. 13). *Gurus Respond—Are We Artists?* [Online]. Available: http://www.code-muse.com/blog/?p=20.

[8] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas (2001). *The Agile Manifesto*. [Online]. Available: http://www.agilemanifesto.org.

[9] Jeff Atwood (2007, Sep. 26). *Can Your Team Pass The Elevator Test.* [Online]. Available: http://www.codinghorror.com/blog/archives/000962.html.