Agile Record

The Magazine for Agile Developers and Agile Testers

October 2010
issue 4

www.agilerecord.com free digital version made in Germany

Developing Software Development

by Markus Gärtner

The software business resides in a constant crisis. This crisis has already lasted since the sixties, and every decade since then seems to have had an answer to it. Among the most popular and most recent movements were the Software Engineering and the Agile movement. In his book *Software Craftsmanship - The New Imperative* [1] Pete McBreen argues against the engineering metaphor and explains why it just holds for very large or very small projects, but not for the majority, the medium sized software development projects.

As Albert Einstein said, *"We cannot solve our problems with the same thinking we used when we created them"*. So far, every aspect of the software crisis turned out to be self inflicted in order to sell training or educational courses on the solution that happened to be mainstream at the time. Since *essentially, all models are wrong, but some are useful* (George Box), this article will take a closer look at the useful aspects of the latest answers to the software crisis, software engineering and craftsmanship.

To avoid any confusion, the term software development in this article will mean programming, testing, documenting and delivery. Similarly, a software developer may be a programmer as well as a tester, a technical writer or a release manager. I will provide a compelling view on the overall development process and compare it to the terms we may have adapted from similar models like Software Engineering or Software Craftsmanship.

From Software Engineering...

Engineering consists of many trade offs. For example, an engineer developing a car makes several trade offs:

- fuel consumption vs. horse power
- horse power vs. final price
- engine size vs. car weight.

An engineer considers these variables when constructing a car and uses a trade off decision to achieve a certain goal that the car manufacturer would like to reach. Thereby he will ensure that the car is safe enough given the time he has to develop the car.

Software programmers as well as software testers also deal with trade offs in their daily work. For example, a software tester considers the cost of automation and the value of exploration. The more time the tester spends on automating tests, the less time there is for exploring the product. The more time is spent on exploration, the less time will be available to automate regression tests for later re use. Figure 1 illustrates this trade off.

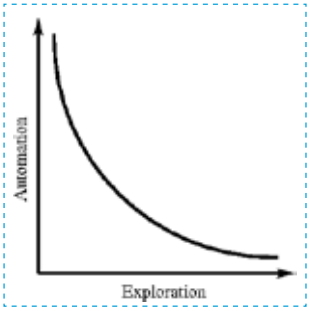

Figure 1: The exploration vs. automation trade off in software testing

The level of automated testing constitutes another trade off decision. Automating a test at a high system level comes with the risk of reduced stability due to many dependencies in the surrounding code. Automating the same test at a lower unit level may not cover inter module integration problems or violated contracts between two modules. Figure 2 shows this trade off.

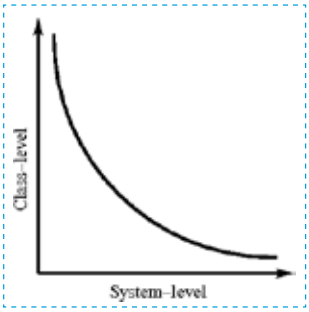

Figure 2: The composition decomposition trade off in software testing

Similarly, there are four such trade offs mentioned in the Agile Manifesto. The last sentence makes them explicit:

"That is, while there is value in the items on the right, we value the items on the left more."

Using the same graphical representation as before, figures 3(a) 3(d) illustrate the values from the Agile Manifesto:

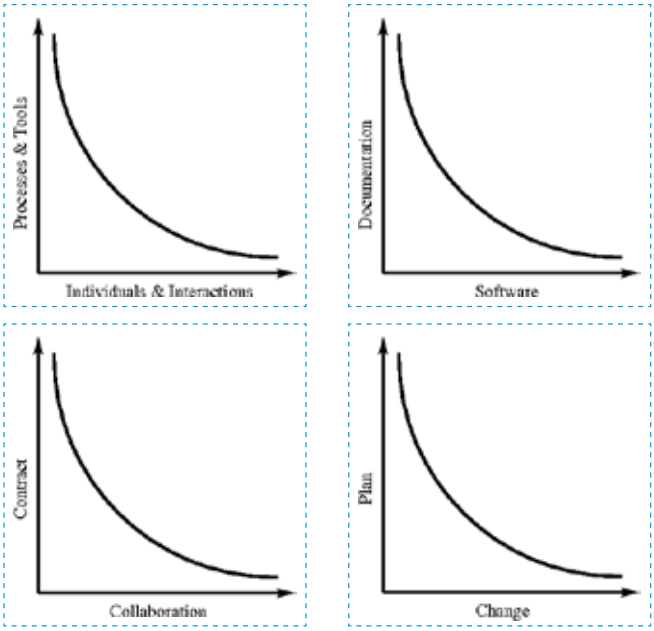

Figure 3: The four Agile value statements as trade offs

At times a software project calls for more documentation. The project members by then are better off spending more time on documentation and less time on creating the software, thereby creating less software. Similarly, for a non collaborative customer more time may be spent on negotiating the contract. The trade offs between individuals and interactions as opposed to processes and tools as well as responding to change opposed to following a plan need to be decided for each software project. Agile methods prefer the light-weight decisions to these trade offs, but keep themselves open for heavy weight approaches when project and context call for it.

... towards craftsmanship ...

In his book[1], Pete McBreen describes the facets of craftsmanship by and large. We have to keep in mind, though, that craftsmanship just like engineering provides another model on how software development can work. This model is suitable for understanding the basic principles, but, as with every model, it leaves out essential details resulting in a simplified view on the overall system.

McBreen's main point is that the software engineering metaphor does not provide a way to introduce people new to software development to their work. Therefore he introduces the craft metaphor. The Software Engineering model does not provide an answer on how to teach new junior programmers, testers, technical writers, and delivery managers on their job. And in fact, Prof. Dr. Edsger W. Dijkstra already noticed this in 1988. Back then, Dijkstra wrote an article on the cruelty of really teaching computer science [2]. According to Dijkstra, the engineering metaphor for software development and delivery leaves too much room for misconceptions, since the model lacks essential details.

The craft analogy provides a model for teaching people new to software development on the job, and does so in a collaborative manner by choosing practices to follow, deliberate learning opportunities and providing the proper slack to learn new techniques and practices. All these aspects are crucial to keep the development process vital. Experienced people teach their younger colleagues. The younger colleagues learn how to do software development while working on a project. By taking the lessons learned directly into practice, new and inexperienced workers get to know how to develop software in a particular context. Over time, this approach creates a solid basis for further development in software and as well as personal.

... and beyond

There are other aspects in the craft metaphor, although these ideas, too, had been flowing around since the earlier days of the Software Engineering movement. Taking pride in your daily work, caring for the needs of the customer, and providing the best product within the given time, money and quality considerations that the customer made. Of course, every software development team member is asked to provide their feedback on the feasibility of the product to be created. This includes providing a personal view on the trade offs that each individual makes to estimate the targeted costs and dates.

Software Development

Dijkstra wrote in late 1988 about the cruelty of analogies [2]. Likewise, a few years earlier Frederick P. Brooks discussed the essence and the accidents of past software problems [3]. Brooks stated that he did not expect any major breakthrough in the software world during the ten years between 1986 and 1996 that would improve software development by any order of magnitude. Reflecting back on the 1990s, his point seems to hold to a certain degree.

Since these two pioneers in the field of software development wrote down the prospects of future evolutions, another decade has past. Reflecting on the points they made about a quarter of a century ago, most of them still hold. However, the past ten years of software development with Agile methods, test driven development and exploratory testing approaches show some benefits in practice. What we as a software producing industry need to keep in mind, however, is the fact that software engineering as

well as software craftsmanship are analogies, or merely models. They provide heuristics, and heuristics are fallible. On the other hand, these models provide useful insights that help us understand some fractions of our work. The models focus on a certain aspect of the development process, while leaving out details that may be essential at times but not for the current model in use.

From the engineering metaphor, trade offs are useful. Given the complexity of most software projects, trade offs provide a way to keep the project under control, while still delivering working software. Systems thinking can help to see the dynamics at play to make decisions based on trade offs. From the craft analogy, apprenticeships help to teach people on the job and help them master their skills. Where traditional education systems fail, the appealing of direct cooperation with an apprentice helps to teach people relevant facets of their day to day work.

While the analogies help, we need to keep in mind what Alistair Cockburn found out in his studies on software projects [4]:

- Almost any methodology can be made to work on some projects.
- Any methodology can manage to fail on some projects.

That said, the analogies apply at times. We need to learn when a model or analogy applies in order to solve a specific problem, and when to use another model. No single analogy holds all the time, so finally creating and maintaining a set of analogies is essential for the people in software development projects, in order to communicate and collaborate. ■

References

[1] Software Craftsmanship The New Imperative, Pete McBreen, Addison Wesley, 2001

[2] On the cruelty of really teaching computing science, Prof. Dr. Edsger W. Dijkstra, University of Texas, December 1988

[3] No Silver Bullet Essence and Accidents of Software Engineering, Frederick P. Brooks, Jr., Computer Magazine, April 1987

[4] Characterizing people as non linear, first order components in software development, Alistair Cockburn, Humans and Technology, 1999

> About the author

Markus Gärtner

is a senior software developer for it-agile GmbH in Hamburg, Germany. Personally committed to Agile methods, he believes in continuous improvement in software testing and programming through skills. Markus co-founded the European chapter on Weekend Testing in 2010. He blogs at blog.shino.de and is a black-belt in the Miagi-Do school of software testing.