The nearly forgotten Value of

Abstraction in Software Engineering

Gebhard Greiter, 2011

Abstraction in software engineering was invented by Parnas, was heavily discussed over nearly two decades (up to 1995) and – sadly – is now nearly forgotten (more precisely: It survived in our modern, object-oriented programming languages but is, since we have UML, no longer used outside code).

Because UML is a language to design code structure (i.e. the structure of HOW a system is implemented), we simply forgot that first of all designers need to gain a clear understanding of WHAT the system is to do and that for arriving there it helps a lot to abstract from the HOW.

A system in this sense can be anything – even a process or a software development project.

The fact that software designers have focused on UML and have over UML forgotten to use abstraction may be the reason why the concept of ***Agile*** Software Development is still seen as being defined by the Agile Manifesto, i.e. by specifying HOW software developers are to work to deliver “valuable software”.

Is this bad? And can abstraction help here? My answer to both of these questions is a firm YES. Let me explain this:

It is – without any doubt – true that the classical waterfall models for the software development process cannot cope with the fast moving world of today. Some of the principles statet by the Agile Manifesto however are certainly *not* the right answer to this problem:

In the Agile Manifesto 5 statements and 12 principles speak about HOW to achieve the goal of delivering better software. WHAT better software actually should be is left undefined resp. is mentioned in one sentence only:

Our highest priority is to satisfy the customer
through early and continuous delivery of valuable software.

Just to say that our goal is *early and continuous* delivery of *valuable* software is simply not enough: What exactly is valuable software?

In my point of view *maintainability and quality under various aspects* are mandatory to make software valuable. But can value in this sense really be achieved by following the rules set by the Agile Manifesto? I don't think so.

I know that we need a better, more abstract definition of what it means to become agile – a definition focusing on the goals (i.e. on WHAT exactly defines valuable software). Only then can we start a successful search for better and better ways to produce value in this sense.

Going this way is important because the first way suggested – the way the Agile Manifesto sketches – is definitely not the best one: It is not stressing the fact that when searching for more agile ways to produce software we must not forget sound software engineering principles (especially not the principle of abstraction discussed so deeply in the years between 1975 and 1995 $^{\text{but has then only survived in programming languages}}$).

Another problem with the development process suggested by the Agile Manifesto is that this process may be useful for maintaining software but is certainly problematic where we need to develop software for a predefined fixed price in a quite specific time frame.

If, e.g. software is to be developed to support a specific event (e.g. the Olympic Games in a specific year soon to come) the definition of "valuable software" cannot be the same as it is when we have to develop a product that we hope will be on the market for a very long time.

So we see: The best way to achieve "valuable software" cannot be same in every project.

Another fact is that Agility in the view of a developer is certainly not the same as Agility in the view of an – always necessary – project manager.

Remembering the value of abstraction, I prefer the following (abstract) definitions of Agile:

Software Developer's Definition of Agile:

Agility means to have a process in place that will allow us (and urge us)
to react on changing business requirements as soon as possible
— accept that the goal is a moving target —

Project Manager's Definition of Agile:

Agile Project Management means to have a process in place
that is to maximize team efficiency

Accepting these definitions of Agile, we can analyze the quite serious shortcomings of the development process suggested by the Agile Manifesto, and then find better and better ways to become agile in a way that really is progress and does not forget lessons learned in past: In this way only we can ensure that one step forward (Agility) is not also one step backwards (the process suggested by the Manifesto).

We see: Using abstraction forces us to think goal-oriented and in terms of WHAT we want to achieve. The Manifesto however, because it describes HOW we should work, focuses on *one* specific way to reach this goal and so, by definition, we cannot find better solutions for the development process.

The fact that first Extreme Programming and then the Agile Manifesto

- have been heavily discussed by people all over the world 10 to 15 years already
- but are still not much more than the Manifesto and SCRUM

convince me that the Manifesto is a cul-de-sac: The WHAT (agility) is a step forward, but the HOW specified by the Manifesto is definitely a step backwards; What Gartner found clearly indicates that the development process suggested by the Manifesto is unacceptable because it does not produce software that is easy enough to maintain.

My advice is:

- Let Agile be defined by the two abstract definitions above (and no longer by the Agile Manifesto).
- Let us then see how we can modify the classical development processes to the effect that we can keep their advantages even if becoming Agile in the sense of these new, now purely ***goal-oriented*** definitions.
- Last but not least let us NEVER AGAIN forget the power of abstraction.

A first candidate for the process we want to find is SST (Specify, Subcontract, Test) – the software development process I favor for every project having at least two developers: SST does not sacrifice classical virtues, does not hinder us to be agile, and is designed to help us reaping the fruits of abstraction.

Author's homepage: http://greiterweb.de/spw/

URI of this document: http://greiterweb.de/spw/dox/Abstraction_in_Software_Engineering.pdf

By the way, I am not alone when I think we need a more abstract definition of Agile. In 2006 already John Rusk wrote:

"Agile development is hard to define, because most people define it by giving examples. For instance they give a specific description of Extreme Programming (XP), instead of defining agile development in general. We've ended up with a widespread misconception that agility is about XP techniques like Pair Programming and Test-Driven Development (TDD). …

"Defining agility by describing XP is like defining democracy by describing America. Such a "definition" obscures the underlying concept with details of the chosen example."