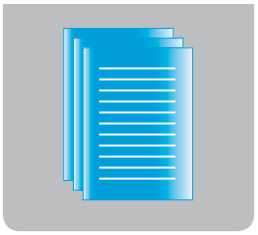
Agile Record

The Magazine for Agile Developers and Agile Testers

www.agilerecord.com free digital version made in Germany ISSN 2191-1320

October 2011

issue 8



Done should mean value delivered to stakeholders

by Tom and Kai Gilb

There is a continuous debate in Agile circles on the meaning of "Done".

In our view the answers offered are unfortunately the same levels of immature and narrow thinking that characterize IT in general, and Agile in particular (4).

The original Agile Manifesto had its heart in the right place. It tried to hint that we preferred to deliver value to customers early and iteratively! *Wonderful*, except, most Agile teachers, gurus, coaches and practitioners do not seem to know what 'value' really means (hint, it does not mean bug free code, functions, or stories, or use cases). And the terms 'customer' and 'user' are far too narrow to encompass the larger set of project and system *stakeholders*, that we are responsible for.

Agilistas did not merely formulate their laudable intent badly. They never taught or practiced the delivery of real value to stakeholders, in the way they should have, to potentially bring credit to our profession. Well, what can you expect from 'coders' who *seem* uninterested in delivering real system value to the real stakeholders. *(Yes we do hope someone feels angry at these cheeky assertions, and who want to prove this wrong, or want to show they have been an exception at value, unknown to us. Good paper for Agile Record.com)*

I fear I must initially define the concept of stakeholder value, since so many Agilistas seem to think it is the same as delivering 'code', use cases, and functions to users. It is not. I will then make the following assertion about the idea of 'Done'.

- 'Done' (for agile projects) means:

  "no more value can be profitably delivered to stakeholders with currently available resources".

So what is this 'value' concept really about?

Here is our formal definition of value, from Competitive Engineering (3)

Value

Value is perceived benefit: that is, the benefit we think we get from something.

Notes:

1. *Value is the potential consequence of system attributes, for one or more stakeholders.*

2. *Value is not linearly related to a system improvement: for example, a small change in an attribute level could add immense perceived value for one group of stakeholders for relatively low cost.*

3. *Value is the perceived usefulness, worth, utility, or importance of a defined system component or system state, for defined stakeholders, under specified conditions.*

An IT system has a set of basic *functions*, defined as *what it does*. These functions tend to exist, and to have existed for the organization or business *independently* of the IT system. A bank lends money, and charges interest. A store takes orders and expedites goods. The reason we build IT systems at all, is NOT to deliver that basic functionality. We do have to *replicate* the functionality, to serve the business or organization *at all*.

So *real* business or organizational *functionality*, has *no value* for a stakeholder. They already *have* it, before the IT system. Banks traded before IT! Movies sold tickets before IT!

There are *other* system attributes we want, when we make IT systems. These are called (in systems engineering) '**performance** attributes'. They include all the *quality* attributes (how *well* the system performs, '-ilities').

I can safely assert that the only reason or justification for any IT system, the only stakeholder values of it, are to be found in the *improved performance characteristics* of the system. How *fast*, how *much*, how *well*, how *costly*. This obvious point seems to have escaped IT nerds' attention.

Some of the nerds are confused conceptually. They think everything they code is a function. But there are in fact two distinct things we program:

- **Functionality**: what the system does
- **Technical Solutions** (aka design, architecture): which result in specific performance levels.

  For example:
  we might code an encryption, to get a level of security,
  we might design and program a screen to get usability,
  we might write tighter code to increase responsiveness,
  and we might 'reduce technical debt' to improve maintainability or portability.

So, some of the code, the code that is *intended* to implement a design, which is *intended* to result in *specific performance improvements*, will result in value to some stakeholders, like *faster, safer, easier, cheaper*.

Let me summarize the ideas up to this point:

1. sufficient 'business' functionality is a minimum price for replacing previous systems, with a consequent view to improving stakeholder value, on that function platform. But, that functionality itself gives no value the stakeholders don't *already* have.

2. IT systems must justify their investment and costs, in terms of the *increased value* they provide to stakeholders, compared to previous alternatives for doing the functions.

Now let's take it one further step. It is not sufficient to *automatically* credit the coding itself, of a design that *intends* to deliver value (improved performance attributes, such as security, to stakeholders). This is the fallacy we have with burn-down charts, or with Scrum sprint velocity alone. We have to be able to quantify and measure, in the real world, the *effect* that this well-intended code and design has had *in practice*. Did it really improve security, usability, maintainability or reduce operational costs in the business? If not, it has no real stakeholder value. And this is not necessarily the fault of the designer or the coder. The value of a design in practice, depends on *many factors* in *addition* to the code. It depends on people, environment, data, other organizations, laws, motivation, training, hardware, networks, etc.

The value of a coded design can be destroyed by any one of these factors alone!

The *consequences* of this fact are:

1. Designers need to consider, and to manage, *all factors* that determine real value generation.

2. This means they *cannot* be software engineers alone, they, or someone else must in fact be a *systems* engineer, or *systems* architect. Code alone does not give sufficient control over value actually delivered.

3. The primary method of delivering any value must be the systems architecture, and follow-up measures of its effects. Programmers cannot do much more than be good sub-suppliers of one of many components of the system.

So. let us conclude:

We software/IT people need to acknowledge that we are not done, until our software component's attributes have successfully helped the system to deliver the stakeholder values that the software was intended to contribute to?

We need to get a lot more professional at consciously defining the necessary software attributes themselves (like security, usability, maintainability), at design engineering the attributes into the software, and at measuring in test, that we have succeed in our own components performance attributes. We are light years away from having this software culture.

But even this capability to *really* engineer reliability, security, usability, maintainability etc. into our software is just one necessary stage in delivering the results to stakeholders that they expect from computer technology; like productivity, cost savings, useful knowledge systems. We as software engineers, need to learn to partner with the overall systems engineering effort to build complete systems. There are serious efforts and practices in this direction (INCOSE.org, (5)) but agile culture does not know about this, care about this, or even reject it explicitly. Unfortunately some areas where agile is being used or explored, and where we work, such as military, health systems, electronics, aviation, and banking are quite serious systems, and need more serious engineering approaches than the agile community has ever tried to offer.

The agile programmers, the Scrum 'Masters' (LOL), and Product Owners are simply not trained, educated or managed to deliver serious stakeholder value. So agile methods, as they stand, must die or change. We suspect Agile Culture is suicidal and would prefer to die out rather than mature, to meet real world challenges.

Iteration, feedback and change (fundamental agile ideas) are powerful concepts for managing software and systems, but right now they are flying blind regarding systematic delivery of software value, which drives systems and stakeholder values. The change is not going to happen through intelligent leadership from programmers. We need intelligent technical management to step up and demand far more from software development.

We are not 'done' until we are considered 'great' at delivering real expected value to stakeholders. We are not even near, are we?

1. Agile Principles Revised -for stakeholder value focus
   http://www.gilb.com/tiki-download_file.php?fileId=431
   Agile Principles in AgileRecord.com, no. 3, 2010

2. Agile Values Revised – for stakeholder value focus
   http://www.gilb.com/tiki-download_file.php?fileId=448
   Agile Values in AgileRecord.com, no. 4, 2010

3. Competitive Engineering, Glossary
   CE Full Glossary
   http://www.gilb.com/tiki-download_file.php?fileId=386

4. http://frank.vanpuffelen.net/2007/08/scrum-utilization-vs-velocity.html
   This is a random example of the narrow mentality that prevails

5. Gilb, Tom, Competitive Engineering, A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, ISBN 0750665076, 2005, Publisher: Elsevier Butterworth-Heinemann.

> About the authors

Tom Gilb and Kai Gilb have, together with many professional friends and clients, personally developed the methods they teach. The methods have been developed over decades of practice all over the world in both small companies and projects, as well as in the largest companies and projects.

Tom Gilb

Tom is the author of nine books, and hundreds of papers on these and related subjects. His latest book 'Competitive Engineering' is a substantial definition of requirements ideas. His ideas on requirements are the acknowledged basis for CMMI level 4 (quantification, as initially developed at IBM from 1980). Tom has guest lectured at universities all over UK, Europe, China, India, USA, Korea – and has been a keynote speaker at dozens of technical conferences internationally.

Kai Gilb

has partnered with Tom in developing these ideas, holding courses and practicing them with clients since 1992. He coach managers and product owners, writes papers, develops the courses, and is writing his own book, 'Evo – Evolutionary Project Management & Product Development.'

Tom & Kai work well as a team, they approach the art of teaching the common methods somewhat differently. Consequently the students benefit from two different styles.

There are very many organizations and individuals who use some or all of their methods. IBM and HP were two early corporate adopters. Recently over 6,000 (and growing) engineers at Intel have adopted the Planguage requirements methods. Ericsson, Nokia and lately Symbian and A Major Mulitnational Finance Group use parts of their methods extensively. Many smaller companies also use the methods.