NEW AGILE PRINCIPLES:
with focus on value delivered to stakeholders

The more often you deploy, the more someone have to wait for someone else. The solution is to make sure the code is always deployable.

Tore Vestues is an architect and developer at BEKK Consulting. He puts his pride into his work and thinks quality must be built into all aspects of software development, from process through code to deploy.

PRINCIPLE 1

Control projects by quantified critical-few, results. 1 Page total !
(not stories, functions, features, use cases, objects, ..)

Most of our so-called functional requirements, are not actually requirements. They are *designs* to meet unarticulated, higher-level, and critical, requirements. For example the requirement to have a 'password' is hiding the real 'security' quality requirement. Most of these really-critical project *quality* objectives are almost always buried in pre-project management slides, and formulated in a woolly and un-testable way ('very robust', 'highly user-friendly'). They are *never* actually used in project architecture, contracting or testing. This is a major cause of project failure. Management and project sponsors are led to believe the project will deliver certain improvements. In practice the agile culture has no mechanism for following up and delivering expected values. Scrum would argue that that is the job of the product owner. But even top Scrum gurus openly acknowledge that the situation in practice is nowhere near what it should be. We simply do not teach and practice the necessary mechanisms. Software people were always bad at this, but agile did not deliver on its' own initial ideals.

PRINCIPLE 2

Make sure those results are business results, not technical.
Align your project with your financial sponsor's interests!

People do not do development projects to get function, features and stories. Yet these seem dominant in the current agile practice. We need functions, stories and perhaps 'features' to make sure the application will do the fundamental business activities that are expected. *Like 'issue an Opera ticket.' 'Give a Child discount.'* But these fundamentals are never the *primary* drivers for investment in a development project. As a rule, the stakeholders already have those functions in place, in current systems. If you look at the project documentation, someone 'sold' management on better systems – some improvements. Faster, cheaper, more reliable etc.

These are usually specifically specified somewhere, and are always quantifiable, as *improvements*. Unfortunately we, in agile development avoid being specific at *this level*. We use adjectives like 'better', 'improved', 'enhanced' and leave it at that. We have learned long ago that our customer is too uneducated, and too stupid (common sense *should* compensate for lack of education) to challenge us on these points. They happily pay us a lot of money for worse systems than they already have.

We need to make it part of our development culture, to carefully analyze business requirements ('save money'), to carefully analyze stakeholder needs ('reduce employee training costs'), to carefully analyze application quality requirements ('vastly better usability'). We need to express these requirements *quantitatively*. We need to systematically derive stakeholder requirements from the business requirements. We need to derive the application quality requirements from the stakeholder requirements. We need then to design, and architect, the systems to deliver the quantified requirement levels, on time. We are nowhere near trying to do this in current conventional agile methods. So we consistently fail the business, and the stakeholders, by not delivering the quality levels required.

Let me be clear here. You can do this *as the system evolves*, and it can be expressed on a single page of quantified top-level requirements. So don't try the 'up front bureaucracy' argument on me!

PRINCIPLE 3

Give developers freedom, to find out how to deliver those results.

The worst scenario I can imagine is when we allow real customers, users, and our own salespeople to dictate 'functions and features' to the developers, carefully disguised as 'customer requirements'. Maybe conveyed by our Product Owners. If you go slightly below the surface, of these false 'requirements' ('means', not 'ends'), you will immediately find that they are not *really* requirements. They are really bad amateur design, for the 'real' requirements – implied

Continues on next page

but not well defined. I gave one example earlier (a real one, Ohio) where 'password' was required, but 'security' (the *real* requirement) was not at all defined.

We are so bad at this, that you can safely assume that almost all so-called requirements are not *real* requirements, they are bad *designs*. All you have to do to see this is ask 'why'? Why 'Password'? (*Security* stupid!) – Oh! Where is the Security requirement? Not there, or worse, stated in management slides as 'State of the Art Security' – and then left to total amateurs (the coders) to design it in!

Imagine if Test Driven Development (TDD) actually tested the *quality* levels, like the 'security' levels, to start with? Far from it; and TDD is another disappointment in the agile kitbag.

In my job analyze real requirements about once a week, internationally, and find very few exceptions – i.e. situations where the real requirements are defined, quantified, and then designed (engineered, architected) towards. Agile culture has no notion of real engineering at all. Softcrafting (coding), sure. But not *engineering* – a totally alien culture.

You cannot design correctly towards a vague requirement ('Better Security'). How do I know if a password is a good design? If the Security requirement is clear and quantified (and I simplify here!) like "Less than 1% chance that expert hackers can penetrate the system within 1 hour of effort", then we can have an intelligent discussion about the 4-digit pin code, that some think is an OK password.

I have one client who pointedly refuses to accept functions and features requirements from any customer or salesperson. They focus on a critical few product qualities (like the usability attribute 'Intuitiveness') and let their developers *engineer* technical solutions, to measurably meet their quantified quality requirements.

This gets the right job (design) done by the right people (developers, not users or customers) towards the right requirements (higher level overall views of the qualities of the application). This client of mine even do their 'refactoring' by iterating towards a set of long-term quality requirements regarding maintainability, and testability. Probably just a coincidence that my client's leaders have real engineering degrees?

PRINCIPLE 4

Estimate the impacts of your designs, on your quantified goals.

I take *quantified* improvement requirements for granted. So do engineers. Agilistas do not seem to have heard of the 'quantified quality' concept. This means they cannot deal with specific, or 'high', quality levels.

The concept of 'design' also seems alien. The only mention of design or architecture in the Agile Manifesto is "*The best architectures, requirements, and designs emerge from self-organizing teams.*" There is some merit in this idea. But, the Agile view on architecture and design is missing most all essential ideas of *real* engineering and architecture.

We have to design and architect with regard to *many* stakeholders, *many* quality and performance objectives, *many* constraints, and *many* conflicting priorities. We have to do so in an ongoing evolutionary sea of changes with regard to all requirements, all stakeholders, all priorities, and all potential architectures. Simply pointing to 'self-organizing teams' is a 'method' falling far short of necessary basic concepts of how to architect and engineer complex, large-scale critical systems..

Any proposed design or architecture must be compared numerically, with estimates, then measurements, of *how well it* meets the multitude of performance and quality requirements. WE must also measure to what degree it eats up resources, or threatens to violate constraints. I recommend my Impact Estimation table as a basic method for doing this numeric comparison of many designs to many requirements. Impact Estimation has been proven consistent with agile ideals and practices, and given far better reported results than other methods (example Confirmit, gilb.com).
If a designer is *unable* to estimate the many impacts of their suggested designs, on our requirements, then the designer is *incompetent*. Most software designers, and software 'architects', are by this definition incompetent. They don't just fail to estimate, they do not even understand their obligation to try!

PRINCIPLE 5

Select designs with the best value impacts for their costs, do them first.

Assuming we find the assertion above, that we should estimate and measure the potential, and real, impacts of designs and architecture on our requirements, to be common sense. Then I would like to argue that our basic method of deciding 'which designs to adopt', should be based on which ones have the best value for money. Scrum, like other methods, focuses narrowly on estimating *effort*. This is not the same as *also* estimating the multiple values *contributed* to the critical project objectives (which 'Impact Estimation' does routinely). It seems strange to me that agile methods understand the *secondary* concept of estimating costs, but never deal with the *primary* concept of estimating value to stakeholders, as defined by their improvement requirements. There is little point in managing cost, if you cannot first manage value. The deeper problem here is probably not Agile methods, but is a total failure of our business schools to teach managers much more than about finance, and nothing about quality and values. If management were awake and balanced, they would demand far more accountability with regard to value delivered by software developers and IT projects. But the development community has long since realized that management was asleep on the job, and lazily taken advantage of it.

PRINCIPLE 6

Decompose the workflow, into weekly (or 2% of budget) time boxes.

At one level the Agilistas and I agree, that dividing up big projects into smaller chunks, of a week or so, is much better than a Waterfall/Big Bang approach.

But I would argue that we need to do more than chunk by 'product owner prioritized requirements'. We need to chunk the *value* flow itself – not just by story/function/use cases. This value chunking is similar to the previous principle of prioritizing the designs of best value/cost. We need to select, next week (next value delivery step to stakeholders) the greatest value we can produce in an arbitrarily small step (our team, working a week). In principle this is what the Scrum Product owner should be doing. But I don't think they are even remotely equipped to do this well. They just do not have the quantified value requirements (above), and the quantified design estimates (above) to make it happen in a logical manner.

PRINCIPLE 7

Change designs, based on quantified value and cost experience of implementation.

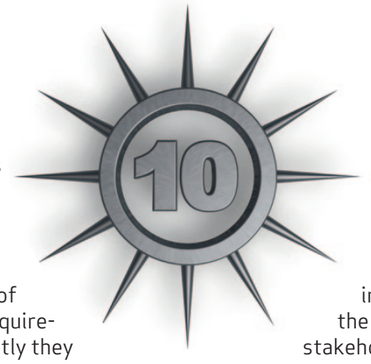
If you get stepwise numeric feedback on the actual delivered value of a design, compared to estimated and perceived value, as is normal at Confirmit, then you will on occasion be disappointed with value achieved. This will give you the opportunity to reconsider your design, or your design implementation, in order to get the value you need, no matter your previous lack of understanding. You might even learn that 'coding alone is not enough' to deliver value to stakeholders.

I fear that this realistic insight possibility is largely lost; since the agile methods neither quantify value required, nor quantify 'value expected' from a step. The result is that we will get stuck with bad designs until it is too late. That does not seem very 'agile' to me.

PRINCIPLE 8

Change the requirements, based on quantified value, cost experience, & new inputs.

Sometimes the quantified quality-and-value requirements are overambitious. It is too easy to dream of great improvement, without being aware of its true cost, or state of the art limitations. Sometimes we have to learn the reality of what we can or should require, by practical *experience*. This is of course normal engineering and science. To learn technical and economic realities step by step.



But the agile community, as we have pointed out, has little concept of quantifying any requirements. Consequently they cannot learn what is realistic. They will just get what they get, by chance or custom.

If they *did* quantify their key requirements, and if they did measure the incremental numeric results, then if requirements were either overambitious, or unacceptably costly, we would have a chance to react quickly (agility).

PRINCIPLE 9

Involve the *stakeholders*, every week, in setting quantified value goals.

Agile methods refer to *users* and *customers*. The terms used are 'sponsors, developers, and users, customers'. In systems engineering (incose.org) there is no doubt that the generic concept is 'stakeholder'. Some parts of software engineering have been adopting a stakeholder paradigm. But agile methods do not mention the concept. In real projects, of moderate size, there are 20 to 40 interesting stakeholder roles worth considering. Stakeholders are sources of critical requirements. Microsoft did not worry enough about a stakeholder called the EU – a costly mistake. Every failed project – and we have far too many – you will find a stakeholder problem at the root. Stakeholders have priorities, and their various requirements have different priorities. We have to keep systematic track of these. Sorry if it requires mental effort. We cannot be lazy and then fail. I doubt if a Scrum Product Owner is trained or equipped to deal with the richness of stakeholders and their needs.

But it can never be a simple matter of analyzing all stakeholders and their needs, and priorities of those needs up front. The fact of actual value delivery on a continuous basis, will change needs and priorities. The external environment of stakeholders (politics, competitors, science, economics) will constantly change their priorities, and indeed even change the fact of who the stakeholders are. So we need to keep some kind of line open to the real world, on a continuous basis. We need to try to sense new prioritized requirements as they emerge, in front of earlier winners. It is not enough to think of requirements as simple functions and use cases. The most critical and pervasive requirements are overall system quality requirements, and it is the numeric levels of the 'ilities' that are critical to adjust, so they are in balance with all other considerations. A tricky business indeed, but – are we going to really be 'agile'? Then we need to be realistic – and current agile methods are not even recognizing the stakeholder concept. Head in the sand, if you ask me!

PRINCIPLE 10

Involve the *stakeholders*, every week, in *actually* using value increments.

Finally – the stakeholders are the ones who should get value delivered incrementally, at every increment of development. I believe that should be the aim of each increment. Not 'delivering working code to customers'. This means you need to recognize exactly which stakeholder type is projected to receive exactly which value improvement, and plan to have these stakeholders, or a useful subset of them, on hand to get the increment, and evaluate the value delivered. Current agile methods are not set up to do this, and in fact do not seem to care at all about value or stakeholders.

In fact developers would have to consider the *whole* system, not just the code, in order to deliver real value – and coders feel very uncomfortable with anything outside their narrow domain.

It is amazing, isn't it, that they have been handed so much power, to screw up society, by 'managers'?