



"turning data into dollars"

Tom's Ten Data Tips – January 2009

Agile Software Development

Agile software development methods are a 'family' of approaches sharing common traits that distinguish it from traditional "waterfall" methods. Waterfall methods begin development by extensive requirements gathering and documentation, and getting "sign-off" on detailed, explicit specifications. "Agile" methods (Scrum, XP, and DSDM being the best known, probably) on the other hand, acknowledge the insurmountable challenge of ever getting the specs *exactly* right. They focus on leveraging creative input from developers and minimizing unnecessary paperwork.

Agile methods represent a paradigm shift from existing software development methods. The imperfect nature and fickleness of requirements is explicitly acknowledged. "People" are put at the center of development, and everything revolves around delivering value for customers as quickly and frequently as possible. Agile is one of the most disciplined methodologies, and strictly adhering to agile practices is required to ensure success.

1. The Industrial age Paradigm Does Not Work For Software Development

As we moved from an era of manufacturing until, say, the early 1900's to the era of mass-production, big strides in manufacturing economy were made. Productivity was raised by breaking down complex and highly skilled tasks into a series of small and relatively simple mechanical tasks (see e.g.: *The Capitalist Philosophers*, Andrea Gabor). This allowed a "dumbing down" of the workforce, to economize on labor costs and make people highly interchangeable. Frederick Winslow Taylor is considered the forefather of this evolution to "scientific management."

Waterfall methods used to work well under this "industrial age" paradigm, and this is probably their most fundamental distinction from agile software development. By translating functional specifications to technical specifications, minute detail is provided on how to deliver the end product. The benefit is that relatively "unconnected" and moderately skilled programmers can jointly create the desired end product.

Now if the requirements would *never* change, this might work reasonably well, albeit at the cost of high upfront investment in documentation. This initial phase is risky and time consuming, and the “architects” writing the documentation form an unavoidable bottleneck in this process. The assumption is that this ‘lost’ initial time can be recouped by spreading the work out over many programmers. The greatest objective risk is having to wait until the end gate before you find out if the product actually ‘works’, and creates value for the business. For progress within the project the *greatest* risks are changing requirements, in particular the effect these have on detailed instructions to the programmers.

2. In Waterfall Methods Requirements Are Fixed, Budget And Time Are Variable

The “standard” approach in a waterfall project is to gather requirements, estimate budget, get sign-off on specs and begin once approved. Management holds the project leader accountable for planning of both budget and timelines. Despite the fact that a timeframe may have been agreed, what happens when the product isn’t finished at the agreed upon delivery date? A big, sunk investment has been made and there are typically few alternatives to pouring in more money and allowing for delays.

The “Agile” approach is to gather requirements in much the same way, but instead of attempting integral planning begin by determining what the most valuable and urgent needs are and start delivering those in (small) chunks. Never mind imperfections, the best way to determine how well the solution “fits” is to get it working in practice as soon as possible. Then in small, rigorously fixed iterative cycles, functionality is appended. Therefore budget and delivery dates are fixed, and the requirements are variable (see also tip# 8). By getting the product *used* as early as possible, closest alignment with customer needs is ensured.

3. Documentation Is Useful Insofar It Gets Used

Agile teams don’t object to documentation. But when they refer to the code as the source of specs, they circumvent the innate nature of specifications to not always be updated even when the code has changed. During refactoring (see also tip# 6) there is ample opportunity to document *inside* the code, which is the appropriate time and place to do so.

Documentation should be produced *only* when it demonstrably adds value to the project objectives. All too often, documenting is very time consuming and gets in the way of delivering value to the business *fast*. Comprehensiveness is the enemy of comprehensibility. Sometimes documentation serves as a 'contract' between buyer and developers which exemplifies the ubiquitous adversarial relation between business and IT.

4. Requirements Are *Always* Subject To Change

One of the recurring complaints from project teams is: "if only they would stop changing the requirements all the time!" There are three reasons why requirements need to change:

- The initial recording from the user was inaccurate
- They were recorded accurately, but as the implications of certain commitments become clear, the customer changes his mind
- The business environment has changed since the requirements were gathered

Probably #3 is used most often to justify repeated changes, and probably illegitimately so. And whether #1 or #2 is the *real* cause for change, it's still better to change now than to find your mismatch *after* delivery of the application, isn't it??

5. Agile Methods Are *Adaptive* Rather Than *Predictive*

The ideas behind traditional methods of software development are to design carefully so that building is predictable and can be planned well. However, if you compare civil engineering with software development, design and build take up very different parts of the project. In civil engineering design/build is, say, 10%-90% whereas in software development that looks more like 50%-50% or even 85%-15% depending on whether you count "testing" as part of building or not.

This principal difference means that "designing" is where you can economize in software projects. It is for this exact reason that agile developers resist "bureaucracy". Non-value added paperwork does not contribute to the end users' needs. On top of that, requirements are accepted as being imperfect. Therefore change needs to be anticipated and embraced! Because the design/build ratio is so different, and requirements are innately unpredictable, *change* needs to drive the method rather than *planning*.

6. Agile Development Without Refactoring Brings You Back To Square One

Common English for refactoring is “cleaning up” code. However, refactoring implies more. It’s ultimate goal is to *keep* the application agile. Refactoring does not remove bugs, nor does it add functionality. It serves to improve maintainability and readability of existing, working code.

So you remove non-functional lines (once needed, later abandoned), you might add commentary, change variable names into more ‘meaningful’ ones, and for instance move “if” statements to a sub routine, or vice versa, depending on what makes the program easier to read.

Because the code is working fine the way it is, there is sometimes temptation to skip refactoring. But remember the very need to work “agile” was to avoid unmaintainable, unchangeable applications. Because refactoring requires ongoing investment as the application is growing, this is one of the great risks of working agile. Skipping this phase or even cutting corners brings you to pretty much the same mess so many of the legacy systems display. Although you may have gotten their quicker...J

7. Agile Methods Are *People Oriented* Rather Than *Process Oriented*

“Waterfall” methods posit a process that provides the path to translate design into the required solution. Along this way, people are interchangeable elements, that don’t necessarily add to the design, they ‘merely’ execute it. It turns out that people physically interacting (preferably using multiple modalities: white board, body language, voice inflection, Q&A, etc.), communicate far more effectively than any formal (paper) process can match.

In an “agile” project, developers play a central role in “creating” the design, in close interaction with (adapting to) the programming choices being made. A technical lead aims to amplify his fellow developers by transferring as much of his skills as possible. As a process, it serves to enable democratic and ego-less programming.

8. An Iteration Must Get *Finished*

The essence of “Agile” is short iterative cycles (typically 1-4 weeks). *Every* iteration must come to completion, so it is eminently importantly

that the allocation of project goals to Must, Should, Could (& Won't – MoSCoW) be set so that musts will *always* be achieved, even in the face of setbacks. "Complete" here means that the new software is tested, documented and integrated in the product under construction so far. It need not be polished nor refactored, but it must *work*. Multiple iterations can sum up to a "release", which can be passed on to a beta test group or (outside) customers.

9. "Design" Means Something 'Else' In Agile Software Development

Agile is sometimes (erroneously) related to evolutionary design, which means that the design *emerges* as the product is built. The larger a system, the riskier this becomes. In particular in light of future change(s). Adjustments deteriorate the system, bugs become ever more vicious until nobody dares to touch it anymore. This is the situation that many legacy applications are in, and we must bend over backwards to avoid that.

The Agile approach to upfront design is that it should enable developers to keep changing the software in the long term. Good design is manifested in the ability to make changes effectively. In practice, it is near impossible to think through all the design challenges upfront. If the design is too rigid, it will force developers to "work around it", thus creating code that tends to become harder to maintain and alter. Hence the emphasis on design as you build. People who do the coding are closest to the actual problems, and are therefore generally best equipped for this task.

10. "Agile" Expands The View On The Value Chain

Because of exceedingly fast iterative cycles and close proximity between developers and (ambassadors for) business users, there is much tighter coupling between design and application *across the value chain*. Working code triggers new as well as better requirements from the end-users. Full time presence of (non-technical) business users in "Agile" projects fosters alignment between organizational needs and IT capabilities. All this serves to move away from the "industrial paradigm" mind set to knowledge work in the 21st century.

A very special thanks to Marjie Carmen for her terrific insights and help in creating this issue of Tom's Ten Data Tips.