Microsoft

Microsoft® Visual Studio®

Team System

Tools for Agility

White Paper

Kent Beck, Three Rivers Institute

June 2008

www.microsoft.com/teamsystem;

msdn.microsoft.com/process

msdn.microsoft.com/practices

CONTENTS

Introduction..1

The Short Form...2

Flow 3

Transparency..6

Tool-Practice Map..8

Predictions..10

About the Author 12

INTRODUCTION

Microsoft has invited me to share my thoughts on the relationship between tools and agile software development. Agile development seeks to increase the value of software development by increasing the feedback available to customers and developers. Some of the feedback comes from more frequent releases to production, some from more frequent testing, some from more frequent software builds, and some from social structures that encourage conversation and dialog. A decrease in cycle time implies an increase in the number of transitions between activities, though, which changes the requirements for effective development tools.

The Agile Manifesto [Beck et al 2001][1] says, "We value processes and tools, but we value individuals and interactions more." Like many attempts to encourage change, this is stated strongly enough that it is open to misinterpretation. Some have taken it to mean that agile software development doesn't require tools, or that the agile development community is populated by neo-Luddites[2] tossing tool CDs onto bonfires and scratching project plans on cave walls with the burnt ends of sticks. I appreciate the opportunity to counteract this impression, to make the case for the appropriate use of tools (and processes) in agile development, and to look forward to the evolution of software development tools.

[1] http://agilemanifesto.org/

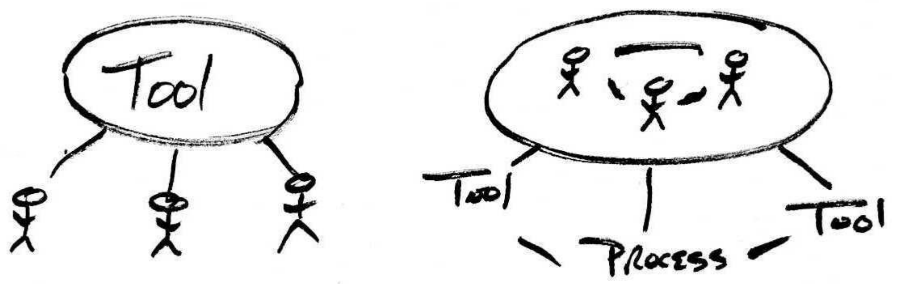
[2] http://en.wikipedia.org/wiki/Luddite

THE SHORT FORM

There is reason and sense behind valuing individuals and interactions over processes and tools. Processes and tools distill common experience. Along comes an uncommon experience and processes and tools can't adapt. Only humans can. That's where the individuals and interactions come in—figuring out how to make progress in unusual situations. A blind adherence to processes and tools in the face of novelty is as ineffective as persisting in variety, debate, negotiation, and creativity in the face of routine and repeating circumstances.

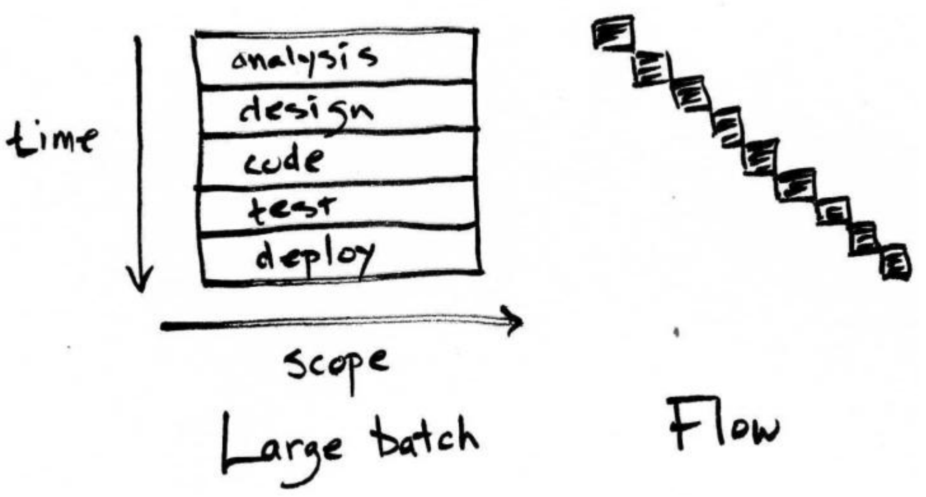
Agile development rests on principles that contradict those found in some software development. The principles of flow, universal responsibility for quality, accountability, and transparency are a few principles of agile development that can lead to substantial changes in development style. Another key principle of agile development is acknowledging that software is done by people. One (seemingly unconscious) metaphor for a software process is that it is like a program, with developers acting as computers. If only you could write the perfect process and support it with tools to enforce that process, goes the thinking, it wouldn't matter how talented, experienced, disciplined, or creative the people on your team, you would be guaranteed success. This bias, the "process as program/human as computer" metaphor, shows through in the emphasis on *repeatability* in the Capability-Maturity Model. In reaction to this, the nascent agile software development community emphasizes the novelty of much software development.

When writing or running code for the first time, no repetition is possible, hence there is no leverage for tools or processes. Even in novel situations, though, some routine remains. Code is tested, written, integrated, and deployed. Even if I've never written a medical digital assistant before, I still remember how to program. I should standardize and automate the tasks that remain routine. If deployment is different than what is supported by my automated deployment tool I may need to deploy manually until I recognize the new routine. I shouldn't twist the architecture to match the tools.



Tools have evolved to efficiently support the separate activities of software development. Agile development, an outgrowth of the nearly-universal drive to ever shorter release cycles, changes the basis of competition for software tools. Rather than single-activity efficiency, tools need to support frequent transitions between activities.

FLOW

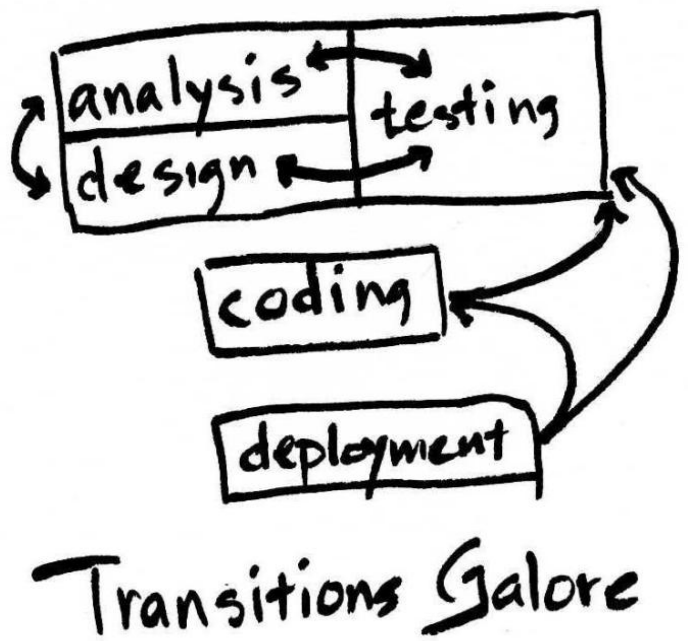


The principle of flow states that, all other things being equal, it is more valuable to deliver smaller batches of functionality more frequently than bigger batches of functionality less frequently. One part of this effect is the ability to react more quickly to change. If the path from new idea to deployed feature is one month long rather than one year, the team can deliver responses to new or changed needs sooner. Another aspect is the greater opportunities for learning that come from sooner and more frequent deployment. Finally, frequent deployment encourages (that so much nicer a term than "forces") the team to learn how to do everything well. A six week manual testing phase makes no sense in a one month delivery cycle.

Flow is valuable for all software development. One way of looking at agile software development is that it pushes the principle of flow further faster. Going from annual releases to quarterly releases may be a strain, but as the trend continues those quarterly releases become monthly, weekly, and even (eventually) daily. Every kind of software is delivered more frequently today than it was ten years ago and the trend will continue indefinitely.

One consequence of increased flow, noted by my partner Cynthia Andres and profoundly affecting tools for agile development, is that the greater the flow, the more transitions between activities. The total percentage of time spent analyzing or designing is likely to be similar (although it would be interesting to verify this), but instead of one transition between analysis decisions and design decisions, flow-oriented development can have tens or hundreds or thousands. The priority for tools shifts from supporting the efficiency of a given activity to efficient switching between activities.

The simplified picture above under-represents this problem of transitions. Rather than a miniature waterfall, agile development mingles analysis and testing, design and testing, coding and test execution, and sets up a quick feedback loop between information gained during implementation and subsequent analysis, design, and implementation decisions.



Tools need to take transitions into account. A user interface that assumes hours of coding at a time followed by a big build process is likely to be cumbersome if applied to a development style requiring multiple build/test cycles per minute. Similarly, a planning tool assuming a major change to plan details every few months is unlikely to smoothly support weekly planning. It's not that these tools couldn't support more transitions, but without constant tumbling in the rushing river bottom of use their transition overhead is unlikely to get polished away.

Teams looking to apply the principle of flow need tools that support rapid transitions between activities. For example, using cards on a wall for planning has many disadvantages—the cards can't be easily distributed to multiple locations, they are impermanent, and spatial information is easily lost. However, the one thing well supported by cards on a wall is transitions. A pair can be in the middle of coding(/analyzing/designing/testing), discover a new requirement out of scope for the current cycle, jot it on a card, and get back to coding without interrupting their development flow—elapsed time ten seconds. Even the quickest switch to another application is likely to take long enough to break their concentration.

Sometimes teams can use parts of tools effectively if they pay attention to the cost of transitions. For example, if tasks are added to a plan daily, taking great care to lay all the tasks out beautifully by hand doesn't make sense. Setting up a tension between aesthetics and accuracy is borrowing trouble. You don't want a project manager dropping tasks on the floor to avoid creating crossing lines. Better to settle for a less attractive but more change-friendly format.

To summarize, the greater the flow, the greater the need to support transitions between activities. Straight ahead efficiency takes a back seat to quick and non-interruptive side tasks. If it comes to a choice, the team is better off with a less effective but more transition-capable tool.

As a tool maker I'm not content to force a tradeoff. I'd like to make and use tools that are both functionally superior (at least for the 80/20 subset of features) and prepared to work in small batches. Beyond that, I'd like tools that take care of the new challenges introduced by agile development.

TRANSPARENCY

The previous section is basically retrospective. It talks about supporting existing activities in software development by encouraging fluid transitions between these activities. There is more to tools for agile development than that.

I was taught that every quantitative change of an order of magnitude creates a qualitative change. Thus, the change from the 10 KPH of a horse to the 100 KPH of a car didn’t just result in faster transportation, it (eventually) changed peoples’ attitudes towards transportation, and the role mobility played in their lives.

In going from annual deployments to monthly deployments to daily deployments we encounter two orders of magnitude of quantitative change. Agile software development, then, is going to be qualitatively different. One change that I wrote about in “Test-Driven Development: By Example” is that programmers need to accept primary responsibility for the quality of their work. To do this efficiently requires tool support as well as a change of attitude. The individual transparency provided by developer-written tests is a prerequisite to greater team transparency.

Team transparency becomes essential with agile software development. When the details of plans change daily, everyone needs a way of finding out what everyone else is doing.

When I started programming the weekly status report was sufficient. Here’s what I did this week, here’s what I’m planning to do next week. Press fast forward twice, though, and the weekly status report becomes as quaint as a debate about the relative merits of assembly language and higher level languages. When coordinating the changing plans of a large, distributed group working on a constantly evolving and deploying system you need more frequent updates. At some point (and here comes the qualitative change) everyone would theoretically be spending all their time reporting the progress they would be making if they weren’t spending all their time reporting progress.

One way out of the Reporting Dilemma is to stop explicitly telling people what you are doing. Instead, rely on your tools to infer your intentions from your activities and report that for you.

This sounds rather Big Brother-ish, but I see an important difference. Rather than Orwell’s central controlling power, transparency is a choice you make to offer trustworthiness to you teammates. A transparent team can more cheaply and effectively coordinate their efforts towards shared goals. Acting transparently sends a signal to others that they can trust you. Trust, when realized, reduces the friction of development as people focus more on what they are accomplishing together and less on avoiding blame. Just as TDD allows me to trust my code and do more with it, trust on a team allows them to be more innovative and experimental.

I have been experimenting with forms of transparency for several years. The first level of transparency was being transparent with myself. I monitored my use of JUnit and discovered that during test-driven development I ran tests every minute or two most of the time. I expanded this idea to an open repository of software development activity, DevCreek[3]. Even just being transparent with ourselves provided unexpected benefits. Using our own development data, for example, we discovered that to support our weekly status meeting we were bringing development to a virtual halt for an entire day.

Transparency can be a big personal shift for developers. Our development on JUnit[4] is projected transparently on DevCreek. Oddly, I resisted making JUnit transparent. Upon reflection I realized I was afraid people would think less of it because we spent so little time on development. So far, though, we don't seem to have lost any sales as a result. I just needed to get used to the idea of telling other people exactly what I was doing.

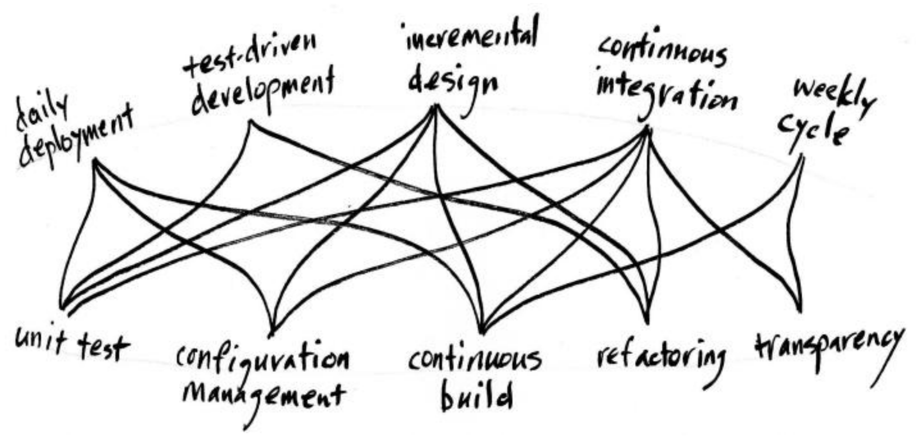
The trend toward greater transparency to wider audiences will continue. It may be hard to unlearn habits and beliefs, but in a world of wide and free flowing information, keeping secrets is a position of weakness. You never know when you are going to be found out. Transparency is the new strength.

[3] http://www.devcreek.com/

[4] http://www.devcreek.com/project/junit

TOOL-PRACTICE MAP

Agile development relies on tools, especially when those tools are tuned for a different rhythm of development. Here is a map of the relationships between some practices common to agile development and some of the tools that support those practices:



Each line in this diagram tells a story. For example, incremental design requires a continuous build tool to quickly smoke out incompatibilities introduced by design changes. Telling all these stories would take more words than I have to share with you here, but what's important to notice is that each tool supports multiple practices, each practice requires multiple tools, and the practices and tools support each other. The strength of this complexity is leverage—a better tool or better way of working can have widespread impact.

It's ridiculous to speak of agile software development without tools. There is so much going on in an agile project every day, so many formerly-manual steps now repeated on fast-forward, that appropriate tools is essential.

Granted, that tooling may not be what is considered state-of-the-art for a slower cycling project. Cards on a wall may be a superior planning and transparency tool, if the computerized alternatives weren't designed for effectively broadcasting frequent changes. Cards on a wall have serious limitations, but they are better than a tool that discourages changes. The challenge for tool makers is to come up with tools that are superior along new dimensions—transition time, change accommodations, transparency. The challenge for tool users is to make effective use of the appropriate subsets of their current tools, make do with low-tech fill-ins when necessary, and communicate their needs clearly to vendors.

This brings us to toolsmithing, a common activity on agile teams. Because the demands placed on tools are often different than what those tools were designed for, teams often spend some of their time adapting or inventing tools. It can be better to invest in a rudimentary tool that supports work than adapt to a tool polished for a different style of work. Over time I expect this balance to shift as more tools appears that are intended to support frequent changes and frequent transitions.

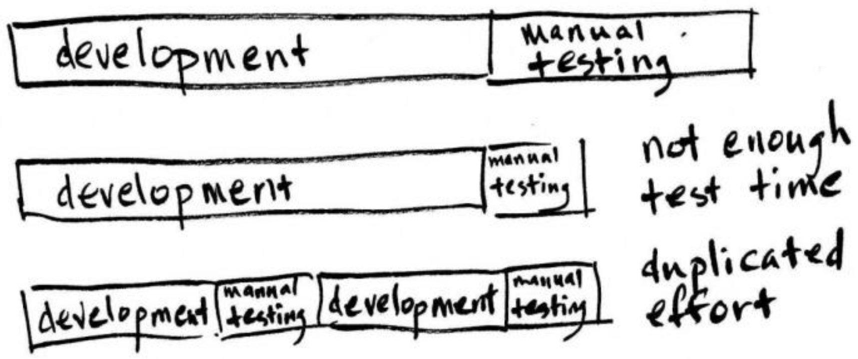
PREDICTIONS

Agile development has already had an impact on developer tools. Unit testing, incremental design, and continuous builds have all arrived in the tool mainstream over the past decade. However, this is only the beginning of the changes. Here are four more changes I see coming:

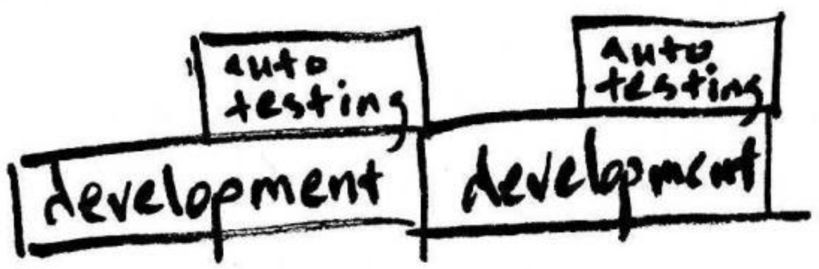
- Smooth transitions between activities
- Greater scope for automated testing
- Transparency
- Real-time collaboration

A trend that will continue to influence software tools is ever-tightening release cycles. Where releases once took years, an increasing number of software products will release new functionality to production monthly, week, daily, or even more frequently. Tools that implicitly assume sequential phases will give way to tools that support parallel (really rapidly alternating) activities. The trend towards support more frequent transitions between activities will continue. More activities will be supported without large changes of context.

Manual post-development testing can only be compressed so far before it loses effectiveness. Manually testing more frequently duplicates effort and risks blunting the focus and attention applied to testing.



A tool-based alternative is to spread the effort of system verification across development with automated tests. As teams gain experience with the new tools and style of work, the additional value of manual post-development testing diminishes. At some point, releasable software results from each round of automated testing:



Currently, only a subset of projects possesses the tools, experience, frameworks, design techniques, and social structures to support such a transformation. Over time, larger, more complex systems will need to move to automated testing and thus frequent releases.

Tools need to simultaneously support rapid change, frequent transitions between activities, and at the same time help the entire team remain focused on large-scale, long-term goals. It is maintaining overall perspective that the kind of transparency discussed earlier becomes valuable. When the cycles become short enough, you don't have time to wait to be told important information. The information needs to be automatically and immediately radiated throughout the team.

In ten years, transparency will be the norm in software development. Rolling up detailed analyses of activities and outcomes will provide developers, managers, and customers to monitor the health of projects and whole organizations. I don't expect this transition to go smoothly. I once asked a room full of programmers at a large company, "What would happen if your software metrics appeared in your annual report?" "*Reported* defects would fall to zero," was the answer. It's understandable that people feel afraid about actively choosing to publish information about programming for the first time, but most of the information that is so jealously guarded is already public knowledge to some degree. The customers already know how many defects you produce—they are the ones who report them.

Finally, here's a prediction from my own experience. I spent 5-10 hours every week pair programming remotely, using screen sharing and a video connection. Whoever hosts the development environment, though, has a big advantage. The delay when I am the remote user is long enough to affect my programming style. I predict that real-time, fine-grained collaboration will become common in tools in the next decade. My explorations along these lines have convinced me that supporting pair programming requires local editing of programs with some way of reconciling collisions when they occur. Once this change is in place, though, I suspect it will trigger further changes as we learn how to support large, distributed teams working transparently to frequently deliver valuable business functionality.

ABOUT THE AUTHOR

Kent Beck is the founder and director of Three Rivers Institute (TRI). His career has combined the practice of software development with reflection, innovation, and communication. His biography can be found at http://www.threeriversinstitute.org/Kent%20Beck.htm