Pragmatic Architecture: Agile Development

Ted Neward

November 2007

Nervously, I look around the room again. It's the weekly meeting again, and everybody's formed into small groups to talk about tonight's topic, "agility". It's another hot buzzword, and once again, I find myself completely clueless about the purpose, practice, or payoff of the whole thing.

"Oh, yeah, we embraced agility a long time ago." While sipping my over sweetened punch, I look at the guy currently claiming the floor in our little group. "After all, if you're not refactoring your code mercilessly, you're just contributing to the problem."

One of the others in the group dares to bare a little ignorance. "I don't understand—without planning ahead, how can you know that you're building something that's not going to have to be ripped out and replaced later?"

As predictable as rain in Seattle, he scoffs at her. "Really. Refactoring is just changing existing code, not rewriting it." He smirks at me. "Sounds like somebody hasn't quite 'gotten' the whole agile thing."

She's not the only one, I think to myself, and move off to find a new group to talk to.

* * *

Somewhere in the last half-decade, how people think about the processes they use to build software changed radically. Dubbed "agile", this new style of developing software hinges not on established documents describing what needs to be built and how people should build it, but on single-sentence descriptions on 3x5 cards, writing lots of code that won't ship as part of the finished product, little planning ahead, and lots of rewriting existing code.

This is supposed to work *how*, exactly? It sounds more like what every young, naïve developer is supposed to want out of a project, and what every old, senior developer has been through before and seen fail. Spectacularly.

We've all heard the horror-story estimates about software development; in fact, it's probably nearly impossible to be a working software developer and *not* hear the depressing statistics. Half of all software projects fail and are cancelled. 80% (or 70%, or 95%, depending on which number sounds right at the time) of all projects are late, over budget, and deliver less than the promised functionality. Developer productivity pales in comparison to the promised productivity of enhancements like functional decomposition, object-orientation, and specific services or components. 105% of all projects are doomed to failure from the beginning due to incompetent users[1]. And so on. Clearly, building software is a high-risk adventure, even more so (at least according to the numbers) than space flight.

The awareness of the success (or failure) ratio in our industry doesn't catch most developers by surprise; most developers find their experience, even after just a few years, echoes the perceived statistics pretty faithfully. And for years, the answer to the problems that caused projects to fail was thought to be rooted in technology: new languages, new platforms, new tools, you name it.

In 1996, Steve McConnell wrote a book, *Rapid Development*, which exposed the fallacy of placing the focus on tools and technology, and put more emphasis on the project management elements of the project. For example, he cited a study conducted in 1984 that found that the adoption of "modern programming practices" could actually *hurt* developer nominal productivity: 0-25% adoption led to an almost 50% reduction in productivity, and 26-75% adoption led to a mean of 0% productivity increase. McConnell then pointed out that the successful software development practice builds on four pillars: classic-mistake avoidance, development fundamentals, risk management, and schedule-oriented practices.

Not surprisingly, most "traditional" software development shops spend most of their time on schedule-oriented practices. Like a basketball team that spends all its practice time focused on dunking the ball and none on passing, set plays, rebounding or free throws, those development teams found their overall success to be sometimes flashy and entertaining, but mostly just one failure after another.

The birth of the agile movement officially came on 13 February 2001, at a ski resort in Snowbird, Utah where 17 individuals, seeking a better way of building software, came together and signed what they called The Agile Manifesto:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while there is value in the items on the right [of each statement], we value the items on the left more.

In other words, focus on those things that matter—the developers, the software, the customer, and the fact that changes will be required no matter how much we might pretend otherwise—rather than those things that traditional software development focuses on—tools, documentation, contracts, and the almighty GANTT chart.

Practically speaking, to the developer, an agile approach centers around maximizing and responding to feedback of all forms, including:

- Customer feedback, based on conversations that are based on their impressions of using the under-construction software, or sometimes based on their changing business needs
- Feedback from the software itself, as evidenced by build failures or exceptions during execution
- Feedback from the developers, based on how they work together (or don't, in some cases)

In essence, rather than trying to faithfully follow the process that led to marginal success on the last project, an agile development approach embraces the fact that every project is different in fundamental ways. Ergo, an agile development team is prepared to adapt to those changes, rather than trying to hammer a way through them as obstacles.

Everything "agile" centers around feedback and reaction to that feedback. Every action in an agile project stresses this notion of feedback. Unit tests are designed to exercise the code so that, in the event of a code change, the team can know immediately if the code accidentally changes its intended functionality. Continuous integration provides immediate feedback if a developer introduces a change that breaks the rest of the project in some way. Pair programming (two developers at one keyboard, coding together) provides a constant feedback loop to each of the developers as they work together to implement a new feature or fix a bug, allowing each to benefit from that feedback—learning something they didn't know before, seeing an old technique applied in a novel way, or in some cases just getting to know that area of the codebase more deeply.

It's not unreasonable to suggest that in an agile approach, the more feedback, the better the project goes.

* * *

Software development isn't the only place where the dichotomy between "big plan up front" and "reacting to feedback" styles becomes evident. Military history tells us that those leaders who were more "in tune" with what was happening on the battlefield, and made decisions based on the changing circumstances there, were far more successful than those who rigidly stuck to the plan[2].

Napoleon, quite possibly the world's most successful general of the modern era, was routinely quoted as saying that he never planned his battles. In fact, one of his greatest battles, Austerlitz, stands as a classic agile-vs-rigid-plan-execution example. His opponents, who outnumbered him by three to one at times, rigidly adhered to the plan they were given by their generals many miles away from the front line. Napoleon, who was known as *le petit corporal* (the little corporal) for his habit of leading from the front lines, split his troops into smaller units called *corps*, gave each to a trusted subordinate who was given a general sense of what Napoleon sought to achieve, and then turned loose.

His *corps*, able to maneuver far more efficiently than the bulky formations of his enemies, routinely found holes in the enemy lines, poured through them, and capitalized on every mistake they found. Metternich, Napoleon's opponent, never had a chance; after having been forcibly retired from the army,

he spent the rest of his life studying and re-studying the battle, trying to figure out what he could have done differently[3].

It's a lesson from which lots of software project managers—and architects—could learn and benefit.

* * *

Not too long ago, one of the Agile Manifesto signatories wrote that the role of the architect no longer served a useful purpose[4]; after all, why bother having an architect if the focus is now to "evolve" a software solution (based on heavy refactoring, iterative feedback, and an unwillingness to implement anything that isn't necessary for today's release) rather than try to "architect" it all ahead of time?

The truth is far more complex and far more subtle.

Most developers have already recognized that there are two wildly different kinds of architects in the world: those who preach and those who practice. The preaching architect rides into town for a project armed with nothing more than Microsoft® PowerPoint®, Microsoft Visio®, and a book of design patterns, establishes a high-level document that somehow intends to provide holy canon for all technical decisions about design and implementation, then rides out again (typically with a rather fat check in her wallet) to cheering crowds and tickertape, having once again rescued a development team from their own ignorance.

Nobody likes preaching architects. Not even their mothers. ("Really, Mom, you could achieve a *much* higher efficiency by cleaning the rooms from the top floor down to the bottom floor, since, as this PowerPoint slide clearly demonstrates, dust falls with gravity, so by cleaning the lower floors after the top floors you capture the dust stirred up from the top floors…")

The problem, of course, is that this approach makes a large number of assumptions that frequently turn out to be wholly incorrect: that the users know what they want, that the business will not change during development, that the tools and technologies available to the project team will not evolve in some way, and so on. The architect simply doesn't have enough context about the business rules, the technology, or the development team to be able to have all the answers up front.

The other kind of architect, the practicing architect, is the architect who, like *le petit corporal*, leads the team from the front lines, evolving the architecture of the project as the project encounters new requirements, changing business scenarios, or new technology. The practicing architect has a level of knowledge that not only spans UML diagrams, but penetrates deep into the nitty-gritty details of programming languages, middleware, databases, and user-interface technologies.

If this sounds like the practicing architect must be deeply knowledgeable in lots of different things, then the message is coming through clearly: simply knowing how to drag-and-drop boxes in Visio or the latest CASE tool doesn't yield a design that will function well. The practicing architect needs to be able to drill into the depths of the system to listen to the feedback being offered by the code itself (running quickly or slowly, scaling well or poorly due to bottlenecks, and so on), and then re-architect as necessary.

In fact, this notion of "re-architecting" is perhaps the most important concept in the agile architect's repertoire. Architecture is filled with decisions that, once made, are impossible (or very difficult) to change. Some of these decisions are outside the architect's ability to influence or change, and simply have to be accepted or worked around. The more decisions that can be made reversible, the better; the one constant element in every software project is change.

Naturally, this comes with a caveat: too much reversibility can lead to too many layers of indirection in a project, leading the software down a path of poor performance because "someday we may want to change something". This is just as dangerous as too little reversibility, and knowing when a project has too little or too much reversibility is what the experienced agile architect brings to the table.

Consider, for example, a classic architectural question: how do we persist the objects we create into the database? To suggest that this can be answered in the span of a single article is, of course, ludicrous, but let's take a shot at it anyway.

As anyone who's studied this problem for five minutes will already know, a variety of options are available. For example, developers can use a direct call-level interface API, such as ADO.NET or JDBC, to issue SQL statements and harvest the results directly. This carries a number of potential problems with it, including verbosity, a tight coupling to the database schema, and a lack of integration with the domain object model the developers will be using. Alternatively, developers can use NHibernate (or some other object/relational-mapping tool) to handle the store/fetch logic internally, and then simply interact with the NHibernate API, which in many ways looks and feels like working with an object database (OODBMS). This approach has its own set of possible problems, liberally discussed elsewhere

[5], and so won't be repeated here. A third approach puts the actual retrieval, storage and any necessary business logic behind a set of stored procedures, called by the developer via a CLI or other tool/library. This approach allows the DBAs to refactor the database schema as necessary, but requires a much deeper commitment to the database vendor, which some IT staffs will find distasteful and risky. A fourth approach suggests that since developers are working with objects, they should just store objects, and use an OODBMS instead of an RDBMS. This obviously makes things easier on the developers, but has huge implications for the rest of the IT infrastructure supporting this application.

Which to choose? Each could be useful in a certain context, and each could be disastrous in a different context.

Here's a fifth option: create a layer of indirection between the developers using the objects and the persistence mechanism. Developers retrieve or store objects through a set of factory methods that hide the actual details, leaving the architect to change (or mix-and-match) those methods as necessary when the context surrounding the project changes. ("Well, after prototyping with db4o during our initial development, we switched to NHibernate once the database schema was finalized. Then we found a couple of queries really needed to be done directly inside the database via a stored proc. Then we discovered a couple of the SQL statements NHibernate generated were pretty hideous and we needed to write them ourselves…")

* * *

For those who haven't figured it out yet, the dirty secret of agility is that agility isn't a new idea—lots of projects have used many of the ideas that are now considered "agile" for years. Developer-authored snippets of code-exercising APIs, once known as "smoke tests", are now called "unit tests." Daily builds are now called "continuous integration." Spiral or iterative development is now called "frequent releases." Code reviews are now called "collaborative development." Getting a buddy to sit with you and help you debug or write something is now called "pair programming." Showing the customer working software on a regular basis is now called "rapid feedback." Rewarding developers for success and keeping them happy is now "valuing the team." And so on.

What hasn't changed, however, is that a significant percentage of the industry still hasn't adopted those practices, whatever their name, that consistently yield better development results. Agile-oriented coaches and presenters cite statistics stating 60% of software projects remain devoid of even one unit test. Another study reports that up to half of software projects still aren't using version control tools.

Call it what you will, the basic keys to successful software remain the same: good people, whether they're developers or managers; good process, whether it's lightweight or rigorous; a good product, whether it's detailed in a document or sketched on 3x5 cards; and good technology, whether it's COBOL or something a bit more modern. Many shops do one thing right, some do two or three, but the truly spectacular successes get all four right, and the results…well, they speak for themselves, and need no buzzword to define them.

[1] OK, so that last one may not be an *official* statistic. But it's still true enough.

[2] As another example beyond the one given here, consider the Battle of the Somme in World War I, as described at http://www.worldwar1.com/sfsomme.htm
[http://www.worldwar1.com/sfsomme.htm] , as a metaphor for the classic "stick to the plan" schedule-oriented style of software development.

[3] *The Campaigns of Napoleon: The Mind and Method of History's Greatest Soldier,* by David G. Chandler, Simon And Schuster, copyright @1966 and 1995

[4] http://www.martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
[http://www.martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf]

[5] The Vietnam of Computer Science [http://www.odbms.org/download/031.01%20Neward%20The%20Vietnam%20of%20Computer%20Science%20June%202006.PDF] , Ted Neward