November 2004

An Introduction to Agile Software Development

by Victor Szalvay, co-founder
Danube Technologies, Inc.
Web site: http://www.danube.com
Web log: http://danube.com/blog/
12011 Bel-Red Rd. Suite 201
Bellevue, WA 98005
(425) 688-0888, ext. 812

Introduction

This paper is an introduction to the Agile school of software development, and is primarily targeted at IT managers and CXOs with an interest in improving development productivity. What is Agile? How can Agile help improve my organization? First, I introduce the two broad schools of thought when it comes to software development: traditional sequential, a.k.a. "the waterfall method", and iterative methods of which Agile is a subset. My objective is to demonstrate the short-comings of the waterfall approach while providing a solution in iterative, and more specifically, Agile methods.

Part I – Shortcomings of Traditional Waterfall Approach

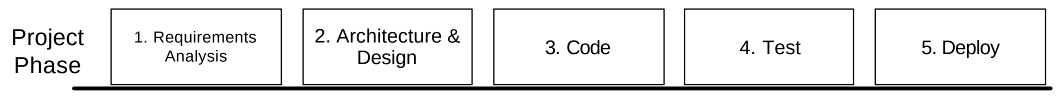
The essence of waterfall software development is that complex software systems can be built in a sequential, phase-wise manner where all of the requirements are gathered at the beginning, all of the design is completed next, and finally the master design is implemented into production quality software. This approach holds that complex systems can be built in a single pass, without going back and revisiting requirements or design ideas in light of changing business or technology conditions. It was first introduced in an article written by Winston Royce in 1970, primarily intended for use in government projects[1].

Waterfall equates software development to a production line conveyor belt. "Requirements analysts" compile the system specifications until they pass the finished requirements specification document to "software designers" who plan the software system and create diagrams documenting how the code should be written. The design diagrams are then passed to the "developers" who implement the code from the design (*See Figure 1*).

Under the waterfall approach, traditional IT managers have made valiant efforts to craft and adhere to large-scale development plans. These plans are typically laid out in advance of development projects using Gantt or PERT charts to map detailed tasks and dependencies for each member of the development group months or years down the line. However, studies of past software projects show that only 9% to 16% are considered on-time and on-budget[2]. In this article, I attempt to summarize current thinking among computer scientists on why waterfall fails in so many cases. I also explore a leading alternative to waterfall: "Agile" methods that focus on incremental and iterative development where requirements, design, implementation, and testing continue throughout the project lifecycle.

Figure 1

Traditional Methods: sequential phased approach



Up-front Requirements Analysis

What are requirements? From the stakeholder's perspective, the requirements are the features and specifications of the system. Requirements define what developers are to build. For example, the system must have a web site with e-commerce capability that can handle 10,000 purchases per hour, or the system must be accessible 99.999% of the time.

One of the biggest problems with waterfall is that it assumes that all project requirements can be accurately gathered at the beginning of the project. In *Figure 1*, the first block represents the requirements analysis phase of a software development project. Analysts slave for weeks or months compiling everything they can gleam about the proposed system into comprehensive "Software Requirements Specification" (SRS) documents. Once finished, the SRS is sent over the fence to the designers while the requirements analysts go to work on the next project.

Imagine a scenario where you engage a software group to build a critical software system. Do you think you could provide every last detail the developers need to know right off the bat? I have yet to encounter such a customer and I am hard pressed to think I ever will. As a start, consider the areas that must be addressed: business rules and exceptions; scalability and concurrent user support; browser or OS support; user roles and restrictions; user interface standards. In fact, it is inevitable that attempts at up-front requirements specification will leave out some very important details simply because the stakeholders cannot tell developers everything about the system at the beginning of the project.[3] This means that the requirements typically change outside of the requirements phase in the form of "change orders", and in many waterfall projects this can be very costly. By virtue of a requirements change, the intricately planned design can be affected dramatically, which will in turn affect any implementation and test strategies. The cost of change in a waterfall project increases exponentially over time because the developer is forced to make any and all project decisions at the beginning of the project.

What if your business needs are still emerging and certain aspects of the system are rapidly changing or cannot be defined yet? Business climates and objectives often change rapidly, especially in today's age of instant information. Can you afford to lock your business into a rigid long-term project where the cost of change grows exponentially? For example, a national test preparation organization commissioned my company to build a simulator for an upcoming standardized test. Since the test itself had not been released yet, the types of questions that would appear on the test were unknown when we started development. But the system had to be done shortly after the tests were released. Markets are forcing the software development community to respond with flexible development plans that flatten the cost of change.

People need to see and feel something before they really know what they want. The "I'll Know it When I See It" (IKIWISI) law says that software development customers can better describe what they really want after seeing and trying working, functional software. I often use a "drawing" analogy to help explain this effect. Although I'm a terrible artist, when I draw a picture I need to see the drawing as I progress. If I tried to close my eyes and draw the same picture, it would prove far less successful. But this is what waterfall asks customers to do: specify the entire system without having a chance to periodically see the progress and make adjustments to the requirements as needed. Waterfall is an "over the fence" approach; the requirements are solicited from the user and some time later the finished product is presented to the user. This is entirely unnatural because customers find it difficult to specify software perfectly without seeing it evolve and progress.

The problem of undefined, changing, and emerging requirements presents a very large challenge to waterfall projects because by definition all requirements must be captured up-front at the risk of costly changes later.

Software Development is more like New Product Development than Manufacturing

Software development is a highly complex field with countless variables impacting the system. All software systems are imperfect because they cannot be built with mathematical or physical certainty. Bridge building relies on physical and mathematical laws. Software development, however, has no laws or clear certainties on which to build. As a result, software is almost always flawed or sub-optimized. Also consider that the building blocks of software projects is usually other software systems (e.g., programming languages, database platforms, etc.), and those systems that act as building blocks contain bugs and cannot be relied on with certainty. Because the foundations of software development are inherently unstable and unreliable, organizations developing software must realize variables exist that are largely outside of management control. It is therefore fair to say that software development is more akin to new product research and development than it is to assembly-line style manufacturing. Software development is innovation, discovery, and artistry; each foray into a development project presents new and difficult challenges that cannot be overcome with one-size-fits-all, cookie-cutter solutions[4].

The waterfall methodology assumes that up-front planning is enough to take into account all variables that could impact the development process. In fact, waterfall projects allocate copious effort detailing every possible risk, mitigation plan, and contingency. But is it possible to predict any and all variables that could possibly affect a software project? The empirical answer is "no" considering the limited success of waterfall projects.[5]

Waterfall therefore equates software development to an assembly line; defined processes can be established that, when used sequentially, result in a successful project each time. The first step is X, the second is Y, and the result is always Z. Can research really be relegated to a series of steps that when performed in sequence result in a new product? If this formulaic approach were adequate, medical researchers could simply plug variables into equations to discover new medicines. On the contrary, since the late 1970s product development companies lead by Toyota, Honda, Fujitsu, 3M, HP, Canon, and NEC, supplanted the sequential "Phased Program Planning" (PPP) approach to new product development with a flexible, holistic approach where the traditional phases of development overlap throughout the product lifecycle.[6] The results were a dramatic improvement in cost and development time to market and ultimately lead to the popular rise of "lean development" and "just-in-time manufacturing". Following the lead of Japanese auto makers, in the 1990s sequential, waterfall-style approaches to new product development were effectively abandoned outside the software development industry.[7] But longstanding insistence from IT managers to categorize software development as a straightforward assembly line progression has kept the software industry from evolving to better methods, the benefits of which other new product development industries have been reaping for decades. It's ironic that a cutting edge technology field like software is so far behind more traditional engineering fields in terms of development methods.

Almost no software system is so simple that the development can be entirely scripted from beginning to end. The inherent uncertainty and complexity in all software projects requires an adaptive development plan to cope with uncertainty and a high number of unknown variables.

Part II - Iterative and Agile methods

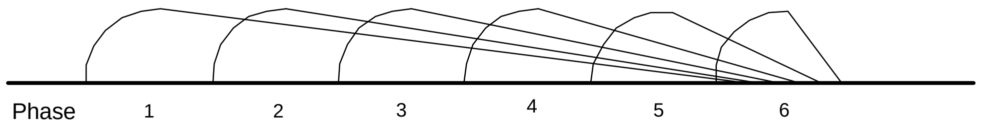
Incremental and Iterative Development

The simple ability to revisit the "phases" of development dramatically improves project efficiency. The idea of revisiting phases over and over is called "incremental and iterative development" (IID). The development lifecycle is cut up into increments or "iterations" and each iteration touches on each of the traditional "phases" of development. For example, with IID requirements is an ongoing process that is periodically revisited. As new requirements surface and as the scope changes, IID processes continually capture the requirements iteration after iteration Interestingly, Winston Royce (of waterfall process fame) later noted that his ideas were incorrectly interpreted and that a "single pass" framework would never work (his article actually advocates at least a second pass).[8] IID allows for multiple "passes", or iterations, over a project lifecycle to properly address complexities and risk factors.

This concept of iterative development hails from the "lean development" era of the 1980s described above where Japanese auto makers made tremendous efficiency and innovation increases simply by removing the phased, sequential approach and implementing an iterative approach, where prototypes were developed for short-term milestones (see Figure 2). Each phase was actually a layer that continued throughout the entire development lifecycle; the requirements, design, and implementation cycle was revisited for each short-term milestone. This "concurrent" development approach created an atmosphere of trial-and-error experimentation and learning that ultimately broke down the status quo and led to efficient innovation.[9] Although direct analogies between industries are never seamless, the success of lean development has influenced a broad class of "iterative" software methods including the Unified Process, Evo, Spiral, and Agile methods.

Figure 2

Iterative approach: Overlapping phases of development



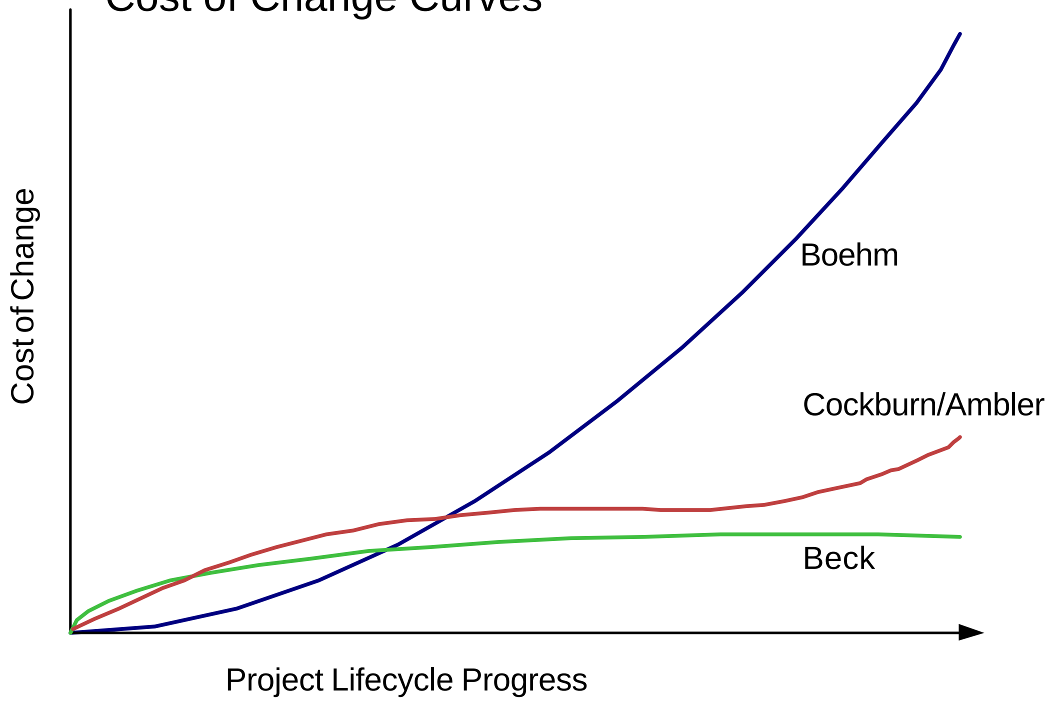
Source: Adapted from H. Takeuchi and I. Nonaka, "The New New Product Development Game", Harvard Business Rev., Jan. 1986, pp. 137-146.

Agile methods: Embracing Change

Agile methods stress productivity and values over heavy-weight process overhead and artifacts. The *Agile Manifesto*[10], a concise summary of Agile values, was written and signed in 2001 although Agile methods have existed since the early 90s. Agile methods promote an iterative mechanism for producing software, and they further increase the iterative nature of the software lifecycle by tightening *design-code-test* loop to at least once a day (if not much more frequently) as opposed to once per iteration. Agile visionary Kent Beck challenged the traditional *cost of change* curve evidenced by Barry Boehm[11] over twenty years ago. Beck's model espouses that the cost of change can be inexpensive even late in the project lifecycle while maintaining or increasing system quality[12]. Beck's idealistic "flat" cost of change curve has since been revised and softened by Alister Cockburn[13] and Scott Ambler[14] to reflect modern corporate realities. Nevertheless, Agile ideals can be applied to reduce the cost of change throughout the software lifecycle even if the cost of change is not perfectly flat.

To accomplish this "flatter" cost of change curve, Agile methods promote a number of engineering practices that enable cost effective change. Author and speaker Martin Fowler describes testing and continuous integration as the "enabling" Agile practices that allow for the advantages gained, like rapid production and minimum up-front design[15]. "Test driven development" is a quality-first approach where developer tests (called *unit tests*) are written prior to the functional code itself. Rather than focusing a lot of effort on big up front design analysis, small increments of functional code are produced according to immediate business need. It is the role of the automated test suite built around the rapidly evolving code to act as a harness that allows developers to make aggressive code changes without fear of undetected regression failure.

Figure 3:
Cost of Change Curves



Object technology and modern integrated development environments (IDEs) boasting built-in testing and refactoring mechanisms negate the expensive Boehm cost of change curve and allow for the cheap change, even late in the project lifecycle.

Agile Project Management: Empirical Process

Scrum, a popular Agile project management method, introduced the concept of *empirical process control* for the management of complex, changing software projects. Scrum holds that straightforward defined processes alone cannot be used to effectively manage complex and dynamic software projects. Risk factors and emerging requirements complicate software development to a point where defined processes fall short. Although it has been attempted in the past, there cannot be a single exhaustive library of defined processes to handle every situation that could possibly surface during a software project. In fact, the manufacturing industry has long known that certain chemical processes, for example, are too difficult to script and define. Instead, an *empirical* or *adaptive* management approach is employed to measure and adjust the chemical process periodically to achieve the desired outcome.[16] As a result, in the Scrum process, project plans are continuously inspected and adapted based on the empirical reality of the project.

Agile project management approaches balance the four variables in software development while keeping in mind the limits associated with new product development. In software development there are four broad control factors. These factors are interconnected, when one changes at least one other factor must also change.

- *Cost* – or Effort. Available money impacts the amount of effort put into the system.
- *Schedule* – A software project is impacted as the timeline is changed.
- *Requirements* – The scope of the work that needs to be done can be increased or decreased to affect the project.
- *Quality* – Cut corners by reducing quality.[17]

Because software development is often considered a sequential, linear process, middle and upper management often assumes that all four of these factors could be dictated to the development team under the waterfall approach. However software development cannot be described by a simple linear process because it cannot be predicted accurately in advance. It is therefore unreasonable to assume that management can control all four of these factors. In reality, management can pick values for three of the four factors at most, and the development process dictates the fourth.[18] The highly complex and uncertain nature of software development makes this expectation of full control unrealistic.

Lean Thinking

Another effective way to analyze how Agile methods increase efficiencies is to apply *lean manufacturing* principles to software development. Although cross-industry analysis can be tenuous, Agile methods have their conceptual roots in the Japanese manufacturing productivity boom of the 1980s[19].

Consider for example the "small batch" principle: things produced in smaller batches are of higher quality and efficiency because the feedback loop is short; controls can be adjusted more frequently, and resources are utilized efficiently to avoid "queuing" (see "queuing theory" and the theory of constraints). Second, Agile methods encourage delaying irreversible decisions until the last responsible moment. Many software development organizations that implement Agile software development are finding they get something they never expected: options. Rather than locking into decisions at the beginning of a project, organizations can reduce risks by leaving options open to decide at a better time when more accurate information is available. Third, the concept of frequent or continuous integration keep software development teams synchronized. Teams can work independently for a while but the code base never diverges for long periods of time, thereby reducing the risks associated with large integrations at the tail end of projects.[20]

Agile Requirements: A Focus on Business ROI

Agile projects avoid "up-front" requirements gathering for the reasons stated above: customers cannot effectively produce all requirements in high enough detail for implementation to occur at the beginning of a project. Customers may not want to make decisions about the system until they have more information. Agile values a high visibility and customer involvement. The frequent demonstration and release of software common in Agile approaches gives customers a chance to "try software" periodically and provide feedback. Agile helps companies produce the "right product". An iterative approach allows customers to delay decisions as well. Decisions can be delayed to some future iteration when better information or technology is available to optimize the choice. For example, we recently delayed selecting a database package for an application because some of the desired features were not available at that time in the options we had to choose from. We therefore built the system in a database independent manner, and (luckily) a few weeks before the product launch a new version was released by one of the database vendors that solved our problem.

One of the biggest advantages to IID is that work can begin before all of the requirements are known. Many organizations are not fully staffed with business analysts cranking out reams of requirements specs. Quite the contrary, in our experience often the bottleneck in the development process has been the lack of availability of customer domain experts for detailed requirements analysis. This is especially the case with small businesses where domain experts wear many hats and often cannot commit to two or three months of straight requirements analysis. IID is ideally suited then to take on bite-sized chunks of requirements that the customer can easily digest.

How do Agile projects prioritize work? A study by the Standish Group shows that in typical software systems 45 percent of the features are never actually used by users and another 19% are only rare used.[21] This is largely because the unused features were specified in some up-front plan before the ratio of their cost to value was considered or even knowable. Focusing on high business value features first is therefore a critical component of efficient Agile development. Because we can change direction rapidly (every iteration) and the cost of change is low, there is a valuable opportunity for the customer to re-examine business factors at the beginning of each iteration to select features for inclusion in the current iteration according to business ROI. Of course, the development team must bring technical risks to the customer, but in the end it is the customer that decides what the development team builds.

Convergence with Agile

One of the most commonly asked questions by those examining Agile is, "how do you know when the software will be finished if there's no up-front plan?" and the obligatory follow up question, "how can we budget for such a project?" It sounds a bit scary: let's start working in short iterative cycles that yield demonstrable software without actually planning everything in advance. But we already know that we cannot plan for *everything* in advance.

The Agile answer is to examine project progress empirically, rather than trying to guess how things might shape up *a priori*. Agile processes like Scrum and XP use a concept called *velocity,* which is the amount of estimated effort a team can complete in a time-boxed iteration. Once a team has established a velocity, a Project Burndown Chart can be utilized to estimate the eventual conclusion of an estimated backlog of work. Each point of the chart in Figure 4 represents an iteration (or Sprint in Scrum), and the Y-axis represents the total estimated effort remaining for the backlog. As iterations progress, a trendline can be established through the points to create a velocity (work the team can complete per iteration). The trendline can then be extrapolated to determine the X-intercept, which represents the empirical estimate of the completion date.

Figure 4: Product Burndown Chart with Velocity

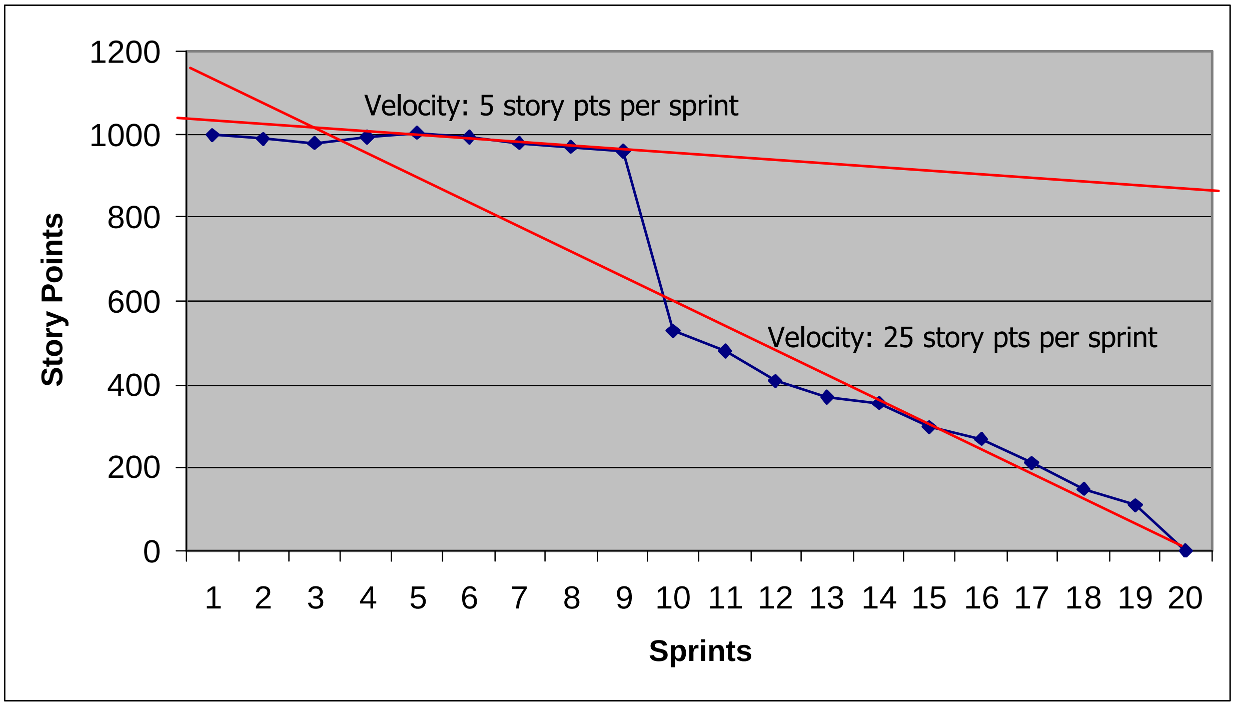


Figure 3 is a Product Burndown Chart representation of a typical project. Through the first nine iterations, the project's "burndown" trendline indicated an X-intercept well into the future (off the charts!). By iteration ten, the product scope had been adjusted down so that the project could be completed by the budgeted 20th iteration. Notice also that the velocity (slope) changed for the better. In this case, perhaps the reduction in scope was accompanied by the removal of critical impediments to efficient progress.

For complex problems like project convergence, Agile methods tell us that the customer cannot specify all four of the software development variables (cost, schedule, scope, quality). To answer the questions above, if the timeline and cost variables are fixed, then the scope of the work must be variable or the definition of the scope must be at a high level so the robustness of each feature can be negotiated.[22] That is, work will proceed on the highest priority requirements first to ensure that the most important things get done before a deadline or the money runs out. Going to production with high priority features is better than never going to production at all, especially considering the Standish report cited above that nearly 65% of features are never or rarely used in reality. And quality is non-negotiable; the features built should always be high quality, adhering to strict code and testing standards. There is no guarantee all features will be built, but it is certain that the highest priority features will go into production and that they will be built well.

Conclusion

But does Agile/IID work? Of course the proof is always in the pudding, and the most recent 2004 Standish Group CHAOS report on the success of software projects shows a dramatic improvement in the failure rate of software projects. In 1994, Standish reported a 31% failure rate that has improved to 15% in 2004.[23] Standish Chairman Jim Johnson attributes the improvement to smaller projects using *iterative* processes as opposed to the waterfall method.[24]

The notion that Agile is a radical deviation from the long established, tried and true history of waterfall software development is incorrect. Although waterfall is often referred to as "traditional", software engineering has had a very short history relative to other engineering disciplines. Unlike bridge building, software development is not built on thousands of years of trial and error, and is therefore in a rapidly evolving infancy as an engineering discipline. Agile is simply the latest theory that is widely replacing the waterfall approach that itself will change and evolve well into the future.[25]

References

After reading an introductory article, sometimes I wish I could ask the author for a list of good books or articles to further my knowledge on the subject. Below are a few good starting points for anyone interested in learning more about agile software development.

Craig Larman, "Agile & Iterative Development: A Manager's Guide", Addison-Wesley, 2003. This book should be read by any manager interested in Agile. It has the most comprehensive empirical evidence for Agile/Iterative of any book currently on the market. It also nicely summarizes and contrasts some of the major Agile/Iterative approaches such as Scrum, XP, and the Unified Process.

Mary Poppendieck, Tom Poppendieck, "Lean Software Development An Agile Toolkit", Addison-Wesley, 2003. This book is outstanding and each page seems to offer a valuable nugget of information. It contains the most compelling case for agile over sequential development I have yet to uncover; much of this article leans on the Poppendieck's work. It also delivers outstanding tools for implementing agile.

Ken Schwaber, Mike Beedle, "Agile Software Development with Scrum", Prentice Hall, 2001. This book is about "Scrum", a highly effective agile project management method. Theory and rhetoric are nice, but how do you do agile? Scrum is a very good place to start on the management side.

Kent Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley, 2000. So what do your developers do differently in agile? Extreme Programming (or XP) advocates engineering principles such as pair programming and test driven development and Beck's book is the *de facto* authority.

[1] Winston Royce, "Managing the Development of Large Software Systems", *Proc. Westcon*, IEEE CS Press, 1970, pp. 328-339.
[2] Standish Group International, Inc., "Chaos Chronicles", 1994, http://www1.standishgroup.com//sample_research/chaos_1994_1.php
[3] Kent Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley, 2000, pp. 18-19.
[4] Ken Schwaber, Mike Beedle, "Agile Software Development with Scrum", Prentice Hall, 2001, pp. 89-94
[5]Standish Group International, Inc., "Chaos Chronicals", 1994, http://www1.standishgroup.com//sample_research/chaos_1994_1.php
[6] I. Nonaka, H. Takeuchi, "The New New Product Development Game", *Harvard Business Review*, January 1986, pp. 137-146.
[7] For more on how lean development influences agile software development, see: Mary Poppendieck, Tom Poppendieck, "Lean Software Development An Agile Toolkit", Addison-Wesley, 2003.
[8] Craig Larman, Victor R. Basili, "Iterative and Incremental Development: A Brief History", *Computer*, IEEE CS Press, June 2004, p. 48.
[9] I. Nonaka, H. Takeuchi, "The New New Product Development Game", *Harvard Business Review*, January 1986, pp. 137-146.
[10] The *Agile Manifesto* is online at http://www.agilemanifesto.org/
[11] Barry Boehm, "Software Engineering Economics", Prentice Hall PTR, 1981.
[12] Kent Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley, 2000.
[13] *Reexamining the Cost of Change Curve, year 2000*, by Alistair Cockburn; XP Magazine, September 2000.
[14] *Examining the Cost of Change Curve*, by Scott Ambler; Agile Modeling Essays excerpted from the book "The Object Primer, 3rd ed.: Agile Model-Driven Development with UML2"; by Scott Ambler, Cambridge University Press, 2004.
[15] Martin Fowler, "Is Design Dead", http://martinfowler.com/articles/designDead.html , 2004.
[16] Ken Schwaber, Mike Beedle, "Agile Software Development with Scrum", Prentice Hall, 2001, pp. 100-101
[17] In fact, it is debatable whether Quality is really an adjustable factor. As professionals, software developers find it very objectionable when asked to skimp on quality. Surgeons or lawyers would be sued for malpractice, and for the same ethical implications software developers resent this charge. Software developers should always aim for high quality software, period.
[18] Kent Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley, 2000, pp. 15-19.
[19] The Scrum Agile method has its roots in the Nonaka-Takeuchi article referenced above.
[20] Mary Poppendieck, Tom Poppendieck, "Lean Software Development An Agile Toolkit", Addison-Wesley, 2003, p 28.
[21] Jim Johnson, "ROI, It's Your Job!", Published Keynote Third International Conference on Extreme Programming, 2002.
[22] Mary Poppendieck, Tom Poppendieck, "Lean Software Development An Agile Toolkit", Addison-Wesley, 2003, p 32.
[23] Standish Group International, Inc., "Chaos Chronicals", 2004.
[24] "Standish: Project Success Rates Improved Over 10 Years", Software Magazine and Weisner Publishing, http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish.
[25] Software architect Michael James has a semi-serious theory that "test-only" development will soon be feasible with the advances in cheap, clustered supercomputing. Developers write only robust tests and supercomputers will write and compile code until all tests pass.