Agile Methods: Fact or Fiction

Matt Ganis
International Business Machines
17 Skyline Drive, Hawthorne, NY 10532
ganis@us.ibm.com

Abstract

Modern businesses and software developers are continually looking for better, more cost-effective ways to develop software. Compared to 40 years ago we have much cheaper/faster computers, more powerful programming languages, better education, and better understanding of the theory of software development. We also have a number of different software processes/methodologies that are believed to be the "best" way to develop software. This paper will explore the Agile software movement and attempt to demonstrate how it can address the need for a flexible mode of developing software allowing teams to create higher quality code while meeting the exceeding demands of our customers.

1.1 History of Software Engineering

The term "software engineering" can be trace it's origin back to a set of historical conferences hosted by NATO in the late 1960's. In the fall of 1968 and again the in fall of 1969, NATO hosted a conference devoted solely to the subject of software engineering[1]. At the time, the term "software engineering" was not in general use, nor widely accepted. To quote directly from the introduction to the proceedings of the first conference:

> "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering"[1].

For the purposes of this conference, the general attitude was not that software development was actually engineering, but rather, the prevailing assumption was that it would be beneficial to consider software development *as* engineering. As a result, these conferences played a major role in gaining general acceptance for the term and as a result having a profound affect on how programmers produced code. The motivation for these

conferences was that the computer industry at large was having a great deal of trouble in producing large and complex software systems and there was a general feeling of a "software crisis" within the industry.

The essence of this software crisis was that the errors in software systems and the cost of writing that software tended to grow geometrically with the size of a software system[2]. One of the most notorious and well-documented example of the software crisis was experienced by IBM in developing the system software for its third-generation System/360 computers, which was estimated to have a cost of $500 million dollars, and involved over 5,000 person years in its making with a peak development staff of one thousand employees [3].

This notion of engineering software had some serious ramifications on the way in which it was created. Because these programmers had their roots in an engineering discipline, an extreme amount of care was taken when crafting code. Programmers in the late 1950's and into the 1960's were drawn mainly from the engineering field, so it should not be surprising to see that they engineered software like they engineered hardware or a mechanical objects. Programmers practiced hardware concepts as "measure twice, cut once," before running their code on the computer. In a physical engineering discipline this makes sense, since once the raw material is cut or fabricated, an error in measurement meant restarting with new material. As an analogy consider that the cost of computing was rather high [4] so rather than wasting precious computing time, programmers adapted the "measure twice, cut once" rule to programming, where they would double or triple check code before wasting precious computer time on errors.

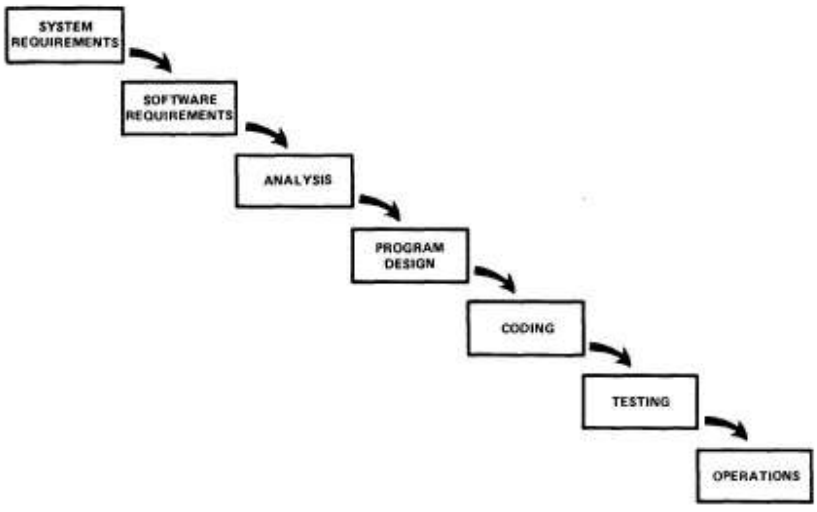

Figure 1. Sample Waterfall Flow for a project

Because of this cautious nature, methodologies were developed that enabled project teams to slowly and methodically create their plans for the creation of software systems. This method, often referred to as the "waterfall" method called for careful, well thought out plans and methods, sacrificing time for detailed specifications and analysis.

1.2 The Waterfall methodology

In the “waterfall method” (see Figure 1) there is minimal feedback from one phase to another (for example from “Coding” to “Testing”). There is often only a small set of artifacts (also called "work products," which can include documents, models, or code) that is produced in each phase, validated at the end of the phase, and then used as input for the next phase. These artifacts are considered complete, almost frozen, and revisited only to fix a major issue. What the waterfall method emphasized predominately was the freezing of requirement specifications or the high-level design very early in the lifecycle, prior to engaging in more thorough design and implementation work.

1.3 A Chink in the Armor

By the later part of the 1970’s, however, people were finding out that working with software differed from working with hardware in significant ways. Software was much easier to modify than was hardware, and unlike hardware, software didn’t require expensive production lines to make duplicates. A programmer would change a program once, and then reload the same code onto another computer, rather than having to individually change the configuration of each copy of the hardware. This ease of modification led many people and organizations to adopt a “code and fix” approach to software development[5], as compared to the exhaustive Critical Design Reviews that are typical in a waterfall methodology.

1.3.1 Cost of Computing

Other factors that led to the “code and fix” mentality (or “spaghetti coders”) included the declining year-to-year cost of computing. Contrary to the early days of “desk checking” code, programmers would opt for the easy way out, and let the computer determine there was error rather than human logic.

Moore's Law (named after Gordon Moore, a founder of Intel) stated that the processing power of microchips doubles every 18 months[6]. This “law” has a corollary that states: as a result of Moore’s law, processor performance will double every 18 months and the cost of computing will drop by nearly 25 percent per year [4, 7]. In 1987, Fred Brooks noted that computing devices saw six orders of magnitude in performance price gain over the previous 30 years [8].

By the end of the 1970’s problems were arising with the formality and sequential nature of the “waterfall processes”. Formal methods had difficulties with scalability and usability by the majority of less expert programmer. In 1975 a survey found that the average coder in 14 large organizations had only two years of college education and two years of software experience [9]. These same coders were familiar with only two programming languages and had limited experience with the fledgling technology. The sequential waterfall model, which was heavily document-intensive, slow-paced, and expensive to use simply created sloppy code [9].

Since much of this planning documentation preceded coding, it was reported that many impatient managers would rush their teams into coding with only minimal effort in requirements and design [9]. In a 1979 survey results indicated that about 50% of the respondents were not using good software requirements and design practices [5]. This resulted in the fact that organizations software costs were beginning to exceed the cost of hardware on which it ran.

In 1972, Barry Boehm predicted that by 1985, more than 80% of the cost of software would be spent in maintenance [10]. A look at trends of percentage of software maintenance compared to overall project cost, show that this prediction is just as true today as it was in 1972 [11].

Year	Proportion of Software Costs
1979	67%
1981	>50%
1984	65-75%
1988	60-70%
1993	75%
2000	>90%

Table 1. Rising costs of software maintenance [19]

Customer or Stakeholder frustration is also associated with the Waterfall method. Since requirements are frozen early in the planning cycles, changes to requirements were difficult if not impossible to inject. Making matters worse, the customer rarely saw "work in progress", having to wait until the end of the development or testing phases before seeing a working product (which may or may not have met their needs).

1.4 Enter Agile Methods

Agile Methodologies have started to gain considerable interest in the IT community during the last several years [12-14]. In his work "The Coming of a New Organization" Peter Drucker discussed the need to change the command-and-control organization to an information-based organization, which is an organization of knowledge specialists [15]. Information-based organizations require clear, simple, common objectives that translate into particular actions. It also needs concentration on one objective or at most on a few.

Because the members of an information-based organization are specialists, they cannot be told how to do their work in a very precise, prescribed way. The heavy handed planning and analysis of the prior methods simply got in the way of making rapid progress. In hindsight, the Agile movement is directly in-line with Drucker's "New Organization". These methods have been proposed as a way to build quality software systems quicker than traditional methods would allow, while easily adapting to a rapidly and frequently changing set of requirements.

Agile development aims to build software faster and more flexibly than traditional approaches. In the Agile manifesto, published in February of 2001, it clearly states that Agile enthusiasts value "Working Software over Comprehensive Documentation" [16]. One of the main tenents of Agile development is the continual production of working software (not complete software, but working versions or prototypes, intended to generate discussion between the developers and the stakeholders). Agile methods are in direct conflict with traditional plan-driven methods of software development that believes that complex software systems can be built in a sequential, phase-wise manner where all of the requirements are gathered at the beginning, followed by design, and finally coding. These plan driven methods assume 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.

In February of 2001, in an effort to address the challenges faced by software developers an initial group of 17 methodologists formed the Agile Software Development Alliance. This group of people defined a manifesto for encouraging better ways of developing software, and then based on that manifesto formulated a collection of principles which defines the criteria for the Agile software development processes. The manifesto defines four values and twelve principles which form the foundation of the agile movement.

To quote from the Agile Manefesto [16]:

"We are uncovering better ways of developing software by doing it and helping others do it. We 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

Agile processes are focused on frequent production of working code through the fast iterations and small increments. Iterative development is an approach to software development where the lifecycle of the project is composed of several iterations that are operated in sequence. These iterations are self-contained projects; consisting of all of the traditional functions that are spread out in the plan-driven methods such as: requirement

analysis, coding and testing. The length of an iteration typically varies from one to six weeks. These timeboxed iterations (with their adaptive and evolutionary refinement of the code and plan) are at the heart of agile methods [17]. The processes are characterized by intensive communication between participants, rapid feedback, simple code design and frequent testing throughout the iteration.

These methods stress productivity and values over heavy-weight process overhead and the production of artifacts such as upfront detailed design documents that are valued by the previous plan driven methods. Agile methods promote an iterative mechanism for producing software, and they further increase the iterative nature of the software by employing such practices as continuous code integration which requires new code to continually be integrated into the larger system, allowing for a working version to be available to the stakeholder or team at a moments notice.

1.5 Scrum

Scrum (named for the scrum in Rugby that denotes "getting an out-of-play ball back into the game") is a popular Agile method that utilizes a high level of teamwork to, was initially developed by Ken Schwaber and Jeff Sutherland, with later collaborations with Mike Beedle[18]. Scrum provides a project management framework that focuses development into 30-day Sprint cycles in which a specified set of Backlog features (system requirements) are delivered. The core practice in Scrum is the use of daily 15-minute team meetings for coordination and integration.

The Approach as defined by Schwaber was developed specifically for the management of a development process. It is an empirical approach to software development that reinforces the ideas of flexibility, adaptability and productivity. If the process or method needs adjustment during a development cycle, changes to the process and team-workings are made "in-flight". While Scrum doesn't define a particular software development technique, it does concentrate on how the members of a software development team should function in order to produce a flexible set of software in an environment where the system requirements are constantly changing.

The important thing about Scrum is that it forces incremental action which creates a need for stakeholder dialog and project feedback throughout the development lifecycle. Referring to Figure 2, in Scrum, the stakeholder[1] input is captured in a feature list called the Backlog. Each month, the development team starts at the top of the Backlog and selects as many of the top priority features as they think can develop in 30 days (the typical length of a Sprint, which is similar to a mini-release). The team then works diligently for a month, completing all of the high priority features (or stories) at which

[1] In Agile, the Stakeholder is assumed to be requester of the product

time the result is demonstrated to the stakeholders for feedback. This provides a basis for rethinking the backlog features and priorities. The stakeholders are allowed to modify and reprioritize the backlog, after which the team selects its next month's work from the top of the list and the process repeats.

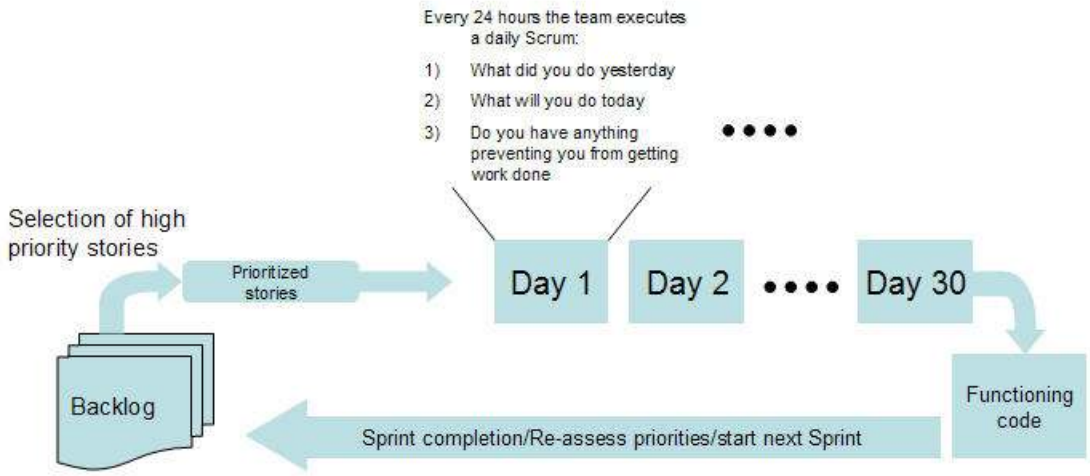

Figure 2. Diagram of a traditional Scrum Sprint

Agile methods stress a high degree of communication between the team members. In Figure 1 you will notice that everyday the team holds a short, 15 minute status meeting called the "Daily Scrum". In this meeting team members relate what they are currently working on, what they plan to do that day and highlight any "blockers" or issues preventing them from completing work. By bring these items into the forefront and uncovering any issues or misconceptions, teams are able to stay well informed of progress of concerns and can quickly address them.

Scrum provides a way for the development team to make regular progress even if the problem is not well understood. At the same time, it provides a method for stake holders to discuss the problem and reach consensus. At the same time, the Scrum process provides a high degree of predictability. Each line on the Backlog is estimated, and the estimates are added to create an overall project completion estimate. After three months, a graph of the Backlog estimate against time is a highly reliable indicator of the project's progress and estimated completion date. Features may be added or subtracted from the Backlog monthly to adjust for constraints as well as changing stakeholder interests. The trend in the Backlog estimate is a reliable indication of whether the project is converging or diverging.

1.6 Extreme Programming (XP)

Extreme Programming (or XP for short) defined by Kent Beck[19] is a lightweight development methodology that is typically employed for small to mediums teams that are developing software in an environment of vague or rapidly changing requirements. The XP methodology asks development teams to follow four core values: communication, simplicity, feedback and courage. These values are used as guidelines to define the 12 practices that comprise the method.

XP provides a set of daily practices that when used together, have been demonstrated to efficiently produce high quality software. These practices include: Whole Team, Planning Game, Customer Tests, Small Releases, Simple Designs, Pair Programming, Test-Driven Development, Recapturing, Continuous Integration, Collective Ownership, Coding Standard and Sustainable Pace. These practices are nothing more than "best practices" as defined in the industry, but when done "to the extreme" and together, can move a development team far ahead of any other non-Agile team. An overview of some of the more "popular" practices include:

1.6.1 Whole team

Many methodologies rely on a divide-and-conquer strategy when attacking a problem. Typically the development process is broken down into distinct steps, different people with different skills are required at each step, and results are communicated from one step to the next through paper documents. However, within an XP team all of the members are involved 100% all the time, and the team members communicate with one another by talking and making each other aware of what the other is doing. This is a very effective strategy, but one that requires all of the team members to stay in constant contact. The preferred method is to have all members of the team sit together in the same physical space. Everyone is kept fully informed, and everyone works together. The time between a question and an accurate answer is kept as close to zero as possible.

1.6.2 Planning Games

Traditional software projects typically focus on the "end date" for a project. Agile projects on the other hand focus on "how much will get done by the due date?" and "what should be done next?" Many methodologies are predictive, making a prediction of what will happen over the course of the project, XP (like most agile methods) is adaptive, continually making changes during the course of the project in an effort to learn from mistakes and emphasize what is working well.

In Release Planning the stakeholder articulates the required features and the programmers estimate the difficulty or sizing. These features are broken down into fine grain "requirements" we call "stories". A story is a small feature that can be easily estimated and implemented during an iteration. Together they lay out a preliminary plan based on the resource available and estimate what can be accomplished with the available people/money/time.

1.6.3 Small Releases

Agile teams are producing working systems within each iteration (or sprint) and as a result they are incrementally improving the system, adding small features or parts of a feature, every day. When complete, they release running, tested software that can be deployed to a production environment at the end of every iteration. This means that the stakeholder can actually use the system and provide feedback to the development team. This is the best way to get high quality feedback on the system.

1.6.4 Simple Design

Because the development team is trying to turn-out systems quickly, they devise the simplest thing that could possibly work, and try to implement it. In Agile, teams will view design and architecture as something done continuously through the course of the development, not something done just at the beginning.

1.6.5 Pair Programming

Pair programming is the practice of having work done by two programmers sitting side-by-side working on the same machine. Intuitively this seems unproductive, however there have reports of higher product quality, better reliability, shorter learning curve for new developers, lower sensitivity to turnover, and shorter time to market and higher job satisfaction of developers [20].

1.6.6 Test-Driven Development

In an Agile environment teams don't add any functionality to a working system unless they have an automated test case to validate that the system doesn't break. This means that developers must write the test for a feature, or an interface, before the code that implements the feature is developed. Test Driven Development ensures that there are tests for every facet of the system, and any developer that makes a change in a system can be confident that there is a test that will tell them if they break something. Through this proactive the cost of change is not only low cost but encouraged.

1.7 Does it work?

But do these methods work?

There have been a number of industry wide surveys that look to see how many enterprises have actually adopted these Agile methods and attempts to quantify their success or failures [21-24]. While there can never be a definitive answer, clearly the evidence points to the fact that, for those that have adopted the methods, there are improvements in a number of areas of their business.

1.7.1 Quality of Software

For any organization, increasing the quality of their product is a major business driver and clearly a desirable result. Table 1 pointed to the increasing cost of software maintenance over time for organizations. Clearly as quality rises, the amount of defects reported (and needing to be resolved) decreases over time. This leads to a higher level of customer satisfaction and a higher profit margins for the Enterprise. In a recent survey by Scott Ambler (Dr. Dobbs journal) he found that for those organizations implementing Agile methods there is a marked increase in the final software product delivered (see Figure 3).

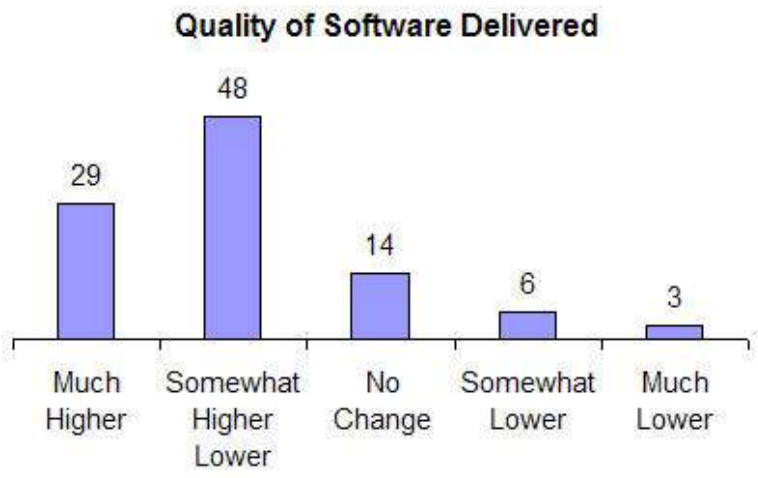

Figure 3. Quality of Agile products

Why do products or applications have an increased level or quality in an agile project? A look at the practices described in section 1.6 helps to shed some light on this. Test Driven Development for example, is an iterative approach to programming where agile software developers must first write a test that fails before they write new functional code (write the test that validates the function, then write the code). There are several advantages of Test Driven Development that Agile teams can realize. With a test suite in hand, agile developers instantly have code available to validate their work, ensuring that they test as often and early as possible in the development lifecycle. Having this test suite also gives the developer confidence to change (or refactor) their code to keep it as well managed and neat as possible. This test suite is critical to the developers in order to detect if they have "broken" anything as the result of their refactoring.

As stated earlier, unlike the waterfall method that would begin its test cycle after the development is complete, Agile methods continually test during the development cycle. This test overload has the result of producing code with little or no defects by the end of the project.

1.7.2 Cost of Producing Software

In Ambler's 2008 Agile Adoption Survey[23], 37% of the respondents reported a lower cost of developing software (and the number soars to 77% if we include no change). Obviously coupling a lower defect product with a lower cost to produce would be a huge benefit to an organization.

How is this possible?

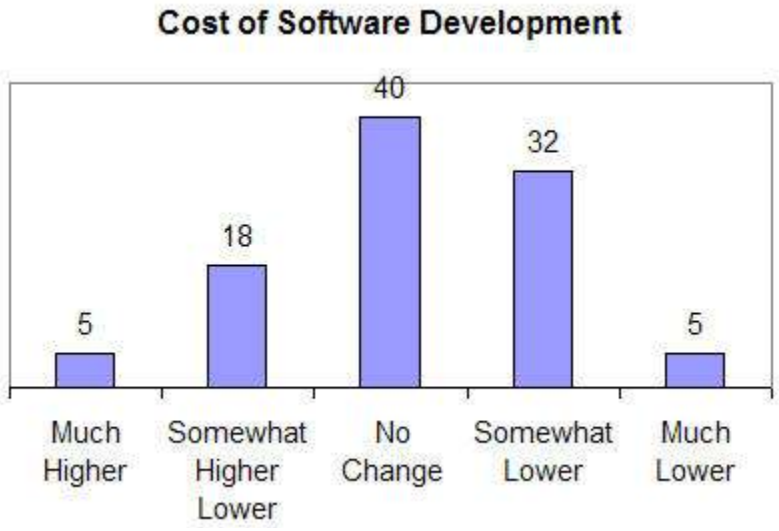

Figure 4. Cost of Agile Products

Referring back to Figure 2, we see that at the end of an iteration (or a Sprint in Scrum), the stakeholder looks at the remaining features to add to the product, prioritizing them for the development team. After seeing a working version of the system under development, its quite possible that the stakeholder would simply accept what has been built at this point as "good enough" and the remaining features could be done in subsequent releases. This has the effect of creating a product at a lower price (by not spending time and money implementing features that clearly don't make a substantial difference to the product) and in less time!

1.7.3 But are they Happy?

Figure 5. Level of Stakeholder Satisfaction

But how happy or satisfied is the stakeholder or requestor of the system? According to Ambler's survey, Agile stakeholders tend to be quite satisfied with the products produced from by Agile teams. The reason for this is quite simple. Again, referring to Figure 2, at the end of every iteration the customer is reviewing the current work product. Not so much for a status, but to provide the development on how well they are understanding the intent of the requested features and to modify features which didn't meet their expectations. The result is that at the end of

the project, there are no "surprises" when the final product is delivered and stakeholders tend to be very happy with the final product.

1.8 Conclusion

As Fred Brooks so eloquently said: "building software will always be hard. There is inherently no silver bullet" [8]. While there is no one answer to the problems we have in the creation of software, there are always pockets of hope. Agile methods, while not perfect, and not the answer for all of our problems, it does go a long way in getting products to market quicker while responding to the ever changing needs of customers. The methods may seem a bit unorthodox, but it's hard to argue with the facts. Agile methods do work; they do produce results and they are here to stay.

About the author:

Dr. Matthew Ganis is a Senior Technical Staff Member (and Certified Scrum master) within the IBM CIO organization. He is the IBM community of practice leader for Agile@IBM, a galvanizing, grass roots effort of over 10,000 IBM Agile practitioners whose aim is to help share methods, stories and advice. He is a member of the Steering committee for New York City's chapter of the APLN (Agile Project Leadership Network) and serves on the editorial board of the International Journal of AGILE AND EXTREME SOFTWARE DEVELOPMENT. He has authored a number of papers/books on his experiences with Agile methods including the soon to be released "Practical Guide to Distributed Scrum" to be published by IBM Press in January of 2010.

Endnotes

1. Naur, P. and B. Randell, eds. *Software Engineering: Report of a conference sponsored by the NATO Science Committe*. 1969, Brussels, Scientific Affairs Division: Garmisch, Germany. 231.

2. Brooks, F., *Mythical Man-month*. 1975: Addison-Wesley. 272.

3. Campbell-Kelly. *Development and Structure of the International Software Industry, 1950-1990*. in *Business History Conference*. 1995.

4. Atkinson, R.D. and R.H. Court, *The New Economy Index - Understanding America's Economic Transformation*, in *Technology, Innovation and New Economy Project*. 1998, The Progressive Policy Institute: Washington, DC.

5. Boehm, B., *Software Engineering.* IEEE Transactions on Computers, 1976. **C-25**(12): p. 1226-1241.

6. Moore, G., *Cramming more Components onto Integrated Circuits.* Electronics, 1965. **38**(8).

7. James, G., *Moore's Corollary and EDA*, in *Electronic Business*. 2003.

8. Brooks, F., *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer, 1987. **20**(4): p. 10-19.

9. Boehm, B. *A View of 20th and 21st Century Software Engineering*. in *ICSE*. 2006. Shanghai, China: ACM.

10. Boehm, B., *Software and its Impact: A Quantitative Assessment.* Datamation, 1973. **19**(5).

11. Vienneau, R., *The Present Value of Software Maintenance.* Journal of Parmetrics, 1995. **15**(1): p. 18-36.

12. Beck, K., *Extreme Programming Explained*. 2005, Boston: Addison-Wesley.

13. Highsmith, J., *Agile Software Development Ecosystems*. 2002: Addison-Wesley.

14. Poppendieck, M. and T. Poppendieck, *Lean Software Development: An Agile Toolkit*. 2003, Addison-Wesley: Boston.

15. Drucker, P., *The Coming of a New Organization.* Harvard Business Review on Knowledge Management, 1998: p. 1-19.

16. Manifesto, T.A., http://agilemanifesto.org/.

17. Larman, C., *Agile and Iterative Development: A Manager's Guide*. 2003: Addison-Wesley.

18. Schwaber, K., M. Beedle, and R. Martin, *Agile Software Development with SCRUM*. 2001, New York: Addison Wesley.

19. Beck, K., *Extreme Programming Explained: Embrace Change*. 1999: Addison-Wesley.

20. Succi, G., et al., *Preliminary Analysis of the Effects of Pair Programming on Job Satisfaction*. Retrieved from: http://www.agilealliance.org/system/article/file/879/file.pdf.

21. Ambler, S., *Agile Adoption Rate Survey: Discussion of the Results.* Retrieved April 4, 2007 from http://www.ambysoft.com/surveys/agileMarch2006.html, 2006.

22. Ambler, S., *Agile Adoption Rate Survey (2007)*, in *Dr. Dobbs Journal*. 2007.

23. Ambler, S., *Agile Adoption Rate Survey: February 2008.* Dr. Dobbs Journal, 2008.

24. VersionOne, *3rd Annual Survey: 2008 "The State of Agile Development"*. 2008: Alpharetta.