

Introduction to agile software development

By Elizabeth Whitworth, elizabethwhitworth@gmail.com

Excerpt from Master's Thesis: *Agile Experience: Communication and Collaboration in Agile Software Development Teams*

Overview

Agile methodologies provide a structure for highly collaborative software development. Developed in the 1990's, the adaptive methodologies were formulated by and for developers in reaction to perceived deficiencies in conventional 'top down' or 'plan driven' methods. Commonly associated with 'lean' engineering (e.g. Poppendieck & Poppendieck, 2003), agile software development closely follows the flow of business value, with a focus on activities that directly contribute to the project end goal of quality software. The agile manifesto (Beck et al., 2001), a guiding force for agile practitioners, was created by 17 influential figures in the nascent software development movement, and is outlined as follows:

Manifesto for Agile Software Development

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, we value the items on the left more.

While the originality and general applicability of the technical and engineering aspects of agile are up for debate (e.g. Boehm, 2002; Larman & Basili, 2003), agile methods are distinctive in that they explicitly integrate behavioral and social concerns into engineering practices. Indeed, the 'people' focus of agile methodologies is singled out as an essential factor in their success and growing popularity (Boehm & Turner, 2004; Cockburn, 2002).

Agile methodologies are specifically designed to facilitate communication, collaboration, and coordination within a dynamic team environment. This includes practices such as shared team rooms to encourage frequent informal communication, informative workplace environments to allow ubiquitous information dissemination, and rapid feedback cycles through which a software product is evolved incrementally in close partnership with a customer representative. Rather than adhering to traditionally long periods of upfront requirements gathering and design before software production, agile teams elicit feedback early on in the process, and deal with the complexities of software development by practicing rapid iterative development from project inception.

Agile practitioners 'fit the process to the project,' restructuring work in reaction to

the needs of the current project, and take pride in rapid flexibility (agility) when faced with the vague and changing needs of today's business context. Development generally occurs in small teams of 10-15 people, who work together as a cohesive whole.

Summary of methods

The following summary of agile is not meant to be exhaustive, with such analysis being found elsewhere (e.g. Abrahamsson, Salo, Ronkainen, & Warsta, 2002; Hayes & Andrews, 2006). The exact definition of agile is elusive, and it is important to note that the term encompasses several different methodologies, including Feature Driven Development (FDD) (Coad, deLuca, & Lefebvre, 1999), Dynamic Systems Development Method (DSDM) (Stapleton, 1997), Crystal (Cockburn, 2005), Agile modeling (Ambler, 2002), Scrum (Schwaber & Beedle, 2002), Adaptive Software Development (Highsmith, 1999), and Extreme Programming (XP) (Auer & Miller, 2001; Beck, 2000, 2004; Jeffries, 2001). The description below focuses on practices associated with Extreme Programming, the most often cited and well-known agile methodology, with reference to other agile literature as found to be relevant to this study.

Building Software

The XP process generally starts with *customer stories*. Customers are instructed to come up with a list of well-defined features or stories that they would like included in the software. This allows *the planning game*, where the development team estimates the time and resources needed to implement each story, and customers use these estimations to select stories they want included in the current iteration (version) of the software based on prioritized business value. Developers begin software production before designing a complete solution, and software is developed in *short iterations*, generally 1-4 weeks in length, at the end of which functional software is completed and tested.

The practice of *self-organizing teams* (Cockburn & Highsmith, 2001; Lindvall et al., 2002) is common in agile. Rigid team structure is avoided, with programmers often able to choose the stories that they would like to work on for the current iteration. Such teams organize themselves in a way that best completes the project at hand, given available resources and the particular capabilities and competencies of the individuals in the team. Full *releases occur often*, generally every 4-6 months, at which point a version of the software can be made available to customers. Agile practices allow *early, concrete, and continuous feedback*, between customer and developers, as well as between the team and the product that is being built. This enables continual negotiation and renegotiation regarding the nature and scope of the software, and how it is to be developed.

An essential XP practice is *test-driven development (TDD)* or *unit testing* (Ambler, 2003, Ch 11; Beck, 2003, 2004; Marick, 2006). TDD is a practice by which code is added in small increments to satisfy simple functional requirements. Before writing any program code, developers write 'tests' that the code must pass. For example, when writing code to support user log in, the tests would concern possibilities in system action, such as what must occur when the user presses enter before entering

anything in the log in box, or what will happen when they enter the wrong password. Developers write tests that verify the correct system response, that is, the tests will pass if the system responds correctly. They first make sure that the tests fail (since the functionality doesn't exist yet), then write only the code necessary to make the test pass.

Test-driven development serves a number of purposes. Firstly, the tests provide a clear focus for developers as they code, with tests and accompanying code often written on a minute-by-minute basis. They also provide feedback on the validity of code once written, similar to the way usability benchmarks are used in interface design. Much of the value of testing comes from the fact that developers accumulate a growing set of '*regression*' tests that can be quickly run to make sure that new functionality does not 'break' any previously written code, i.e. cause it to stop working. Automated tests are used to allow rapid and dynamic interaction and feedback on progress. Tests also provide a means of documentation and communication (Ambler, 2005) in that they effectively summarize the purpose and function of each system module. In addition to programmer tests, there are also *customer acceptance tests*. These are automated tests defined by the customer regarding desired functionality, and are run to ensure that features are implemented correctly (Jeffries, 2001).

Agile processes place value in *simple design*, with a focus on constructing a working product based on the customer stories for the current iteration only. An agile catch phrase declaring "You Ain't Gonna To Need It" (YAGNI) is used to prevent inclusion by anticipation. This puts a focus on building software for today, rather than preparing for possibilities in future iterations. Value is also placed on a *working version of the product*, maintained on a weekly or daily basis. *Incremental, or evolutionary design* (Fowler, 2004) is also practiced. Rather than spending a lot of time planning, analyzing, and designing for a big perfect release at the end of a long development cycle, agile developers do each of these activities, a little at a time, as development progresses (Beck, 2000).

Time-boxing is another common practice used to assist simple design and scheduling. Development activities or projects are split into separate time periods, with a set number of hours allowed for each task. For each period, deadlines and resources are fixed, while deliverables are considered more flexible. Thus the *scope* of a development effort is adjusted to meet scheduling constraints, and functionality that will not fit into the current time period is dropped or reconsidered, usually in negotiation with the customer. At the end of the time box each task should be one hundred percent completed.

In place of defining the finished product at the outset of the project, XP developers utilize *continuous integration and refactoring*. As each programmer completes his or her tasks, they continuously integrate their individual code into the collective system code, often on an hourly basis. Tests are utilized at this point, where a combined suite of unit tests from the entire team must run one hundred percent correctly each time code is integrated into the system. This avoids problems often encountered in non-agile methodologies, where pieces of code can exist and be developed separately for days, weeks, or even months, with disastrous results when attempts are made to integrate them into a single workable 'build' (version of the software system).

Refactoring (Fowler, Beck, Brant, Opdyke, & Roberts, 1999) involves improving the structure of code for simplicity and understandability (and therefore flexibility and changeability), while retaining the same functionality. Basically, software developers review program code to ensure that it is the simplest possible design for the given functionality. If not, they refactor, or revise the code so that it is. This avoids the problems of ‘spaghetti code,’ which is so complex and convoluted as to be indecipherable.

Dependent on code simplicity and readability is the practice of *collective code ownership*. No one person in an XP team ‘owns’ any part of the code, and any pair of programmers can change code anywhere in the system as they see fit. This removes potential bottlenecks and complications in the coding process. In support of this, agile teams often stick to a *coding standard*, whereby all code in the system is written similar manner.

Finally, as mentioned with regards to unit testing, *automation* is extremely important when implementing agile practices (Ambler, 2005). Automated tests and software builds make it quick and easy for team members to see how their code works, both alone and when integrated into the main software system. The ease and speed provided by automation is especially important, since lengthy manual processes are likely to be skipped in the face of time constraints (Jeffries, 2001).

Communication and Collaboration

Agile methodologies put a strong emphasis on constant communication and coordination between team members, particularly face-to-face interaction. *Rapid feedback* is highly valued in the agile communication structure, and lines of communication are kept open and short. If there is any question regarding an aspect of software design that a particular developer is working on, he or she will initiate a short (15 minute) meeting with the person most likely to have the answers that they need. *Stand-up meetings* are a common practice, involving 10-15 minute meetings at the beginning or end of every work day, with each person in the team giving a brief run down on their progress and/or plans for the day.

The customer driven nature of agile is one of the main factors setting it apart from traditional methods (Turner & Boehm, 2003). Agile stresses the importance of a dedicated *on-site customer* representative who “provides the requirements, sets the priorities, and steers the project” (Jeffries, 2001). This can be held in contrast to traditional methods, where interaction with the customer is relatively limited, usually occurring during initial planning and specification phases. The agile customer has much more control over the software development process. Working software is delivered regularly, allowing them to see the product, evaluate the software themselves or with end users, and provide feedback. The quick feedback between the customer and the development team is viewed as a critical success factor in agile projects (Lindvall et al., 2002).

Agile teams operate in a *shared workspace* or “war room,” which basically consists of a large open plan room, with workstations set up so that developers can work in a shared environment. Small cubicles are made available in case team members need privacy to work, but the bulk of activity occurs in the common space.

War rooms facilitate communication and rapid feedback between team members, and have been found to be astoundingly effective in increasing team productivity (Teasley et al., 1996).

Agile teams also operate in *informative workspaces*, which generally involve populating the working environment with *information radiators* to effectively disseminate information between team members (Cockburn, 2002). This could include putting up posters outlining end-user personas, charts specifying project schedules, or elaborate post it note constructions outlining which customer stories have or have not been completed, and by whom. Information radiators are posted on common wall space so that everyone can see them, and are often marked up or changed as the project evolves. The goal of these artifacts is to ensure that everyone is on the same page in their understanding of the project goals, and can quickly see what state the project is in at any point in development (Beck, 2000).

Agile teams also tend to put a lot of emphasis on *interactive communication artifacts* (e.g. Bellin & Suchman Simone, 1997), where discussions and meetings occur around the use of flip charts, white boards, index cards, and post it notes. *Software and code* are also utilized, with a working version of the software acting as a shared artifact around which activity and communication is centered. *Unit tests* can further be used as both requirements specifications before code is written, and documentation of what the code entails after the tests have been satisfied.

Tools created to support agile software development have become increasingly important to team efforts. Tools such as JUnit (Husted and Massol, 2003; Object Mentor, 2006) and Nunit (Hamilton, 2004; Two, Poole, Cansdale, & Feldman, 2005) provide *automated frameworks for writing and managing tests*. Such tools automate what would otherwise be tedious manual processes, and give immediate visual feedback on whether the tests have passed or failed. Testing frameworks such as Framework for Integrated Tests (Fit) (Mugridge & Cunningham, 2005; Shore, 2005) and Fittesse were developed specifically to enhance communication and collaboration with customers.

Using these frameworks, customers (or users) themselves can write acceptance tests into simple, spreadsheet style documents. These tests are then checked against the actual code, allowing the customer, without any knowledge of programming, to interface with the software product directly and in real time. The increased interaction between developers, customers, and code permitted by such tools greatly enhances feedback, code development, and understanding of the software problem on all sides. The resulting records of tests can further be used for documentation purposes.

Finally, we come to *pair programming*; a practice that has garnered much interest in the software engineering discipline, being perhaps the most well examined agile practice to date (Nicolescu & Plummer, 2003; McDowell, Werner, Bullock, & Fernald, 2003; Chong et al., 2005; Hanks, 2003; Hanks, McDowell, Draper, & Krnjajic, 2004; Nosek, 1998). In pair programming developers work in pairs for any coding activity. Pairs usually sit at the same workstation, with one person ‘driving’ at the keyboard and the other sitting beside them ‘navigating’ and providing input. A common practice is for programmers to switch pairs regularly, such as every 90 minutes, or whenever the current task is completed. Along with an extra person

considering and reviewing each coding activity, pair programming offers benefits such as an apprenticeship-like learning environment, and dissemination of domain knowledge, project understanding, and coding skills across the programming team.

On the whole, knowledge dissemination, communication, and collaboration are enhanced through the use of various agile practices and tools that maintain wide and open channels of interaction.

Agile Interactions

A number of themes and ideals permeate agile interactions, helping smooth coordination of the team and development process. *Feedback* has been mentioned as important in both the physical construction of software, and in communication and coordination between the development team. Another role that feedback plays is in *team reflection or retrospectives* (Kerth, 2001). While less common than many of the other practices, agile teams are encouraged to take time at the end of a project or iteration to consider their processes and team interactions and reflect on how they could improve or adapt.

Also mentioned is the idea of *simplicity*, summed up in the agile catchphrase ‘Do The Simplest Thing That Could Possibly Work’ (DTSTTCPW). This principle is applied rigorously to every aspect of the development process, and is used to guard against the very human tendency to put large amounts of time and effort into getting something ‘right’, while adding little value to the desired end goal. This is related to the previously mentioned YAGNI (‘You Ain’t Gonna Need It’), which guards against the tendency to over prepare for later contingencies that may never actually arise.

The main example of this principle can be seen in the use of documentation. In contrast to traditional software development processes, which focus much time and effort into creating and maintaining detailed documentation, agile documentation is maintained only to the extent that it helps the immediate goal of creating software. Note that programmer documentation involves work internal to the software development processes, and should not be confused with user documentation such as usage manuals or help. Agile software development principles, such as DTSTTCPW, focus on simplicity and short-term value to allow for change in the future, and are specifically tailored towards a business context where future requirements are unknown (Boehm, 2002).

Sustainable pace is an XP practice based on the idea that people can work better and longer when they have personal and down time. As such, agile team members are not encouraged towards, and are even discouraged from, working overtime. This is related to a general aversion to unrealistic deadlines and the stress that accompanies it.

Finally, agile teams maintain a *whole team mentality*, and a level of *shared leadership*, where input from team members is encouraged and valued, and each team member has a say in the direction the project is going. Martin Fowler (2005) points out two things that define agile methods. Firstly, agile methods are adaptive rather than predictive. They are designed to deal with change, as opposed to trying to plan large chunks of development under the assumption of no change. Secondly, agile

methods are people, rather than process-oriented. The goal of agile is to define a process that will work well for the team and the project at hand, rather than adhere to a fixed process regardless of other factors.

There are additional aspects of XP and agile software development that deserve mention. Firstly, many agile processes, and XP in particular, are highly integrative methodologies. In other words, specific practices in XP depend on other practices in order for them to be effective. For example, the ability to respond to changes in customer requirements from one iteration to the next is highly dependent on the code simplicity due to refactoring and coding standards. The specific dependencies in the XP process are further outlined by Beck (2000, 2004), but it is worth noting that many agile practices are not standalone procedures.

Also important are the culture and values surrounding the methodologies. Kent Beck (2004) offers the underlying values of courage and respect in addition to communication, simplicity, and feedback, while Cockburn and Highsmith (2001, p.132) state that “agility requires that teams have a common focus, mutual trust, and respect.” An increasingly complex set of values comprise agile ideology and culture, and feed into the experience of individuals interacting within agile environments.

References

- Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. (2002). Agile software development methods: Review and analysis. Espoo, Finland: VT Publications.
- Ambler, S. W. (2002). Agile modeling. New York, NY: John Wiley and Sons.
- Ambler, S. (2003). Agile database techniques: Effective strategies for the agile software developer. Indianapolis, IN: Wiley Publishing Inc.
- Ambler, S. (2005). Quality in an Agile World. *Software Quality Professional*, 7(4), 34-40.
- Auer, K., & Miller, R. (2001). *XP Applied (The XP Series)*. Reading, MA: Addison Wesley.
- Beck, K. (2000). *Extreme Programming explained*. Reading, MA: Addison Wesley.
- Beck, K. (2003). *Test driven development: By example*. Reading, MA: Addison-Wesley.
- Beck, K., with Andres, C. (2004). *Extreme programming explained: Embrace change (2nd ed.)* Reading, MA: Addison-Wesley Professional.
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Martin Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. (2001). *Manifesto for Agile Software Development*. Retrieved March 15th, 2006 from <http://agilemanifesto.org/>

- Bellin, D. & Suchman Simone, S. (1997). *The CRC Card Book (Object Technology Series)*. Reading, MA: Addison-Wesley Professional.
- Boehm, B. (2002). Get ready for agile methods, with care. *IEEE Computer*, 35(1), 64-69.
- Boehm, B., & Turner, R. (2004). *Balancing agility and discipline: A guide for the perplexed*. Reading, MA: Addison-Wesley.
- Chong, J., Plummer, R., Leifer, L., Klemmer, S. R., Eris, O., & Toye, G. (2005). Pair programming: When and why it works. In *Proceedings of the Psychology of Programming Interest Group Workshop*, Brighton, UK.
- Coad, P., deLuca, J., & Lefebvre, E. (1999). *Java modeling in color with UML*. Upper Saddle River, NJ: Prentice Hall.
- Cockburn, A. (2002). *Agile software development*. Reading, MA: Addison-Wesley.
- Cockburn, A. (2005). *Crystal Clear: A Human-Powered Methodology for Small Teams*. Reading, MA: Addison-Wesley.
- Cockburn, A., & Highsmith, J. (2001). Agile software development: The people factor. *IEEE Computer Magazine*, 34(11), 131-133.
- Fowler, M. (2004). Is design dead? In G. Succi, & M. Marchesi (Eds.), *Extreme programming explained*. Boston, MA: Addison-Wesley.
- Fowler, M. (2005). The new methodology. Retrieved January 29th, 2006 from <http://www.martinfowler.com/articles/newMethodology.html>
- Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring - improving the design of existing code*. Reading, MA: Addison-Wesley.
- Hamilton, B. (2004). *NUnit pocket reference*. Sebastopol, CA: O'Reilly Media, Inc.
- Hanks, B. (2003). Empirical Studies of Pair Programming. In *Proceedings of the 2nd International Workshop on Empirical Evaluation of Agile Processes (EEAP 2003)*.
- Hanks, B., McDowell, C., Draper, D., & Krnjajic, M. (2004). Program quality with pair programming in CS1. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*, 176-180.
- Highsmith, J.A. III (1999). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York, NY: Dorset House Publishing Company.
- Husted, T. & Massol, V. (2003). *JUnit in action*. Greenwich, CT: Manning Publications.
- Jeffries, R. (2001). What is extreme programming? Retrieved March 15th, 2006 from <http://www.xprogramming.com/xpmag/whatisxp.htm>

- Kerth, N. L. (2001). Project retrospectives: A handbook for team reviews (2nd ed.). New York, NY: Dorset House Publishing Company.
- Larman, C., & Basili, V. R. (2003). Iterative and incremental development: a brief history. *IEEE Computer*, 36(6), 47-56.
- Lindvall, M., Basili, V. R., Boehm, B., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L., & Zelkowitz, M. (2002). Empirical findings in agile methods. In *Proceedings of Extreme Programming and Agile Methods – XP/Agile Universe Conference 2002*, Chicago, IL, 97-207. http://fc-md.umd.edu/fcmd/Papers/Lindvall_agile_universe_eworkshop.pdf
- Marick, B. (2006). Testing foundations: Consulting in software testing. Retrieved April 15th, 2006 from <http://www.testing.com/>
- McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2003). The impact of pair programming on student performance, perception and persistence. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon. 602-607.
- Mugridge, R., Cunningham, W. (2005). *Fit for Developing Software: Framework for Integrated Tests* (Robert C. Martin Series). Upper Saddle River, NJ: Prentice Hall PTR.
- Nicolescu, R., & Plummer, R. (2003). A pair programming experiment in a large computer course. *Romanian Journal of Information Science and Technology*, 6(1&2), 199-216.
- Nosek, J. T. (1998). The case for collaborative programming. *Communications of the ACM*, 41(3), 105-108.
- Object Mentor, Inc. (2006). JUnit.org. Retrieved March 15th, 2006 from <http://www.junit.org/index.htm>
- Poppendieck, M., & Poppendieck, T. (2003). *Lean software development: An agile toolkit for software development managers*. Reading, MA: Addison Wesley.
- Schwaber, K., & Beedle, M. (2002). *Agile software development with SCRUM*. Upper Saddle River, NJ: Prentice Hall.
- Shore, J., with Woldrich, D. (2005). Introduction to Fit. Retrieved March 15th, 2006 from <http://fit.c2.com/wiki.cgi?IntroductionToFit>
- Stapleton, J. (1997). *DSDM: The method in practice*. Boston, MA: Addison-Wesley Longman Publishing Company.
- Turner R., & Boehm B. (2003). People factors in software management: Lessons from comparing agile and plan-driven methods. *Crosstalk*, 4-8.
- Two, M.C, Poole, C., Cansdale, J. & Feldman, G. (2005). NUnit. Retrieved March 15th, 2006 from <http://www.nunit.org>