The Advantages of Agile Development

John Foley

27 April 2014

Abstract

Agile Methodologies have taken the software engineering industry by storm, especially in this modern era of internet technology and application production. The Agile Manfiesto, published in 2001, introduced many terms and ideas to describe the current set of methodologies that we call Agile. There are many problems in software engineering that Agile Methodologies are designed to address, such as ability to react to change and adapt, maintain communication with clients instead of renegotiating contract, and evolutionary development cycles instead of planned in phases. These solve problems that are inherent in software development, and are not handled well in past methodologies such as Waterfall.

1 Introduction

Agile Methodologies have taken the software engineering industry by storm, especially in this modern era of internet technology and application production. The Agile Manifesto, published in 2001, introduced many terms and ideas to describe the current set of methodologies that we call Agile. Unlike it's predecessors, Agile focuses on non-deterministic, complex systems where its difficult to start designing in the beginning. After years of successes and failures, software developers began seeing that upfront designs of these type of systems only lead to divergence, massive overhauls, and as a consequence produce overwhelming waste. The type of strategy that waterfall for example follows is primarily predictive. Client requirements for the system are determined and processed, and then different phases of design and implementation are strictly followed. Each phase is built on the previous and attempt to predict the future sections of the system that will be addressed in the next phase, until the application is ready to be implemented. This layered style involves traceable documentation and designing every aspect of the application in one long process. Agiles core philosophy is to take the opposing face of the same coin and be adaptive instead of predictive. The same long, phased process that waterfall methodologies advocate are instead broken up into smaller, easier to handle chunks.

Since software development typically takes one to several years, a development methodology must be able to organize and deal with time and money constraints. Agile deals with this problem by introducing iterations, and an iteration is a method of breaking time and features up into workable, adaptive slices. Each series of features or changes are called sprints, each iteration is meant to move the systems development up one step closer to being completed. Each iteration is suppose to be independent enough to not worry about the next iteration, and thus not predict anything because change is actually encouraged in agile methodologies. Change is so expected that agile actually works at its best when the ideas of the application change after each iteration. This aspect makes agile incredibly effective in non-deterministic, complex systems where features are complex and need to be modified with each new addition.

Large systems development requires collaboration between teams and departments in order to complete the product. This leads to another core aspect of Agile, which is the small team approach.

By keeping teams tight and cross functional, communication is abundant and the project is easier to manage. Communication isnt kept exclusive to teammates; clients or client domain representatives are welcomed to be in the same room so that developers can ask questions and maintain convergence with the system that the clients desire. This communication keeps development adaptive and the teams on track with an accurate end product. This flexibility is desired, especially when a business is growing rapidly and requirements may change in a relatively short amount of time [?].

Requirements and constraints are likely to change over time, so a methodology must be able to cope with change over the years of its life cycle and maintenance beyond that. Iterations and constant adaptation leads to the final core philosophy of agile methodologies, evolutionary design [?]. Every iteration organizes and pushes the project up, and can be thought of as a generation. A generation is independent and is expected to be operational by itself, even if it doesnt satisfy every feature and requirements. In fact, it isnt suppose to. Every generation is built on the previous and expected to be better or expanded in some way, and changes can be made easily and quickly.

Procedural methodologies that have been used for years have attempted to cope with these problems in a variety of ways. Waterfall tries to predict and document for every aspect of the application that it can, and does a great job if the project is sequential and any after-the-fact changes are considered prohibitively costly. In a modern era of software development, new methodologies are required to deal with the growing significance of department collaboration, evolutionary design, and long development durations [?]. Agile has been produced to manage these problems.

2 Client Contact with Small, Cross-Functional Teams

Communication with the client during the life of software is critically important. It ensures that the product matches what the client is expecting. In the past, client requirements are constructed into a contract for the developer to create, but that allows for divergence if every detail is ambiguous. Developers couldnt possibly have a complete knowledge of every client domain that they develop for. They couldnt be expected to write code that satisfies every requirement perfectly as each detail requires a small amount of decision making to implement it, and that almost certainly results in slight divergence. Agile methodologies are designed so that the client is in the room with a small group of developers. This way, communication is constant between teammates and the client representative, and any detail or specific implementation can be asked and accurately chosen on the spot and not necessarily what the developer thinks is best.

2.1 Client in the Room

The book in class talks about properties of a wicked problem. A wicked problem does not have a definite formulation, and there is no stopping rule. Every problem is unique and the solver has to be political in the right to implement it their way. This describes more and more problems in this age of software design because every problem is specific to the client and responsive in nature. Interaction with a huge number of variables is common; a website for example interacts with a diverse client set and must communicate with a number of servers or other clients through a variety of protocols. It would be impossible to program software that accounts for every determinable result- instead, software must be diverse and thought about enough to be intelligent in its own wicked way.

Agile attempts to solve this problem by keeping abundant, free communication between developers and clients. The Agile Manifesto values customer collaboration over contract negotiation. This way the business domain that the software should work within is well known, as no one knows it better than the client, and developers are free to continue working knowing that they can accurately address each detail [**?**]. This characteristic of agile methodologies can be seen across the different subtypes.

2.2 Small Teams and Scrums

One of the great challenges of software engineering is that many software development projects require collaboration between multiple departments or teams. Waterfall attempts to address this problem with strict, copious documentation that traces and plans every single step that a developer takes. This allows for all that come after them to follow each decision and then work on that foundation. Of course this works to a certain extent, but pitfalls start to develop when a project starts to be crushed under its own weight [**?**]. If multiple departments start to work on the same pieces of code and then diverge too much, then merging becomes prohibitively tedious.

Agile methodologies attempt to solve this problem through small, cross functional teams that work to solve a very specific set of user stories over a carefully planned duration. These teams communication every day, and one specific methodology to be produced is called scrum. A scrum is a daily meeting between team members where each team member discusses what theyre working on and what they plan on doing. This way each team member is aware of what other team members are doing and inter-team divergence is kept to a minimum [1]. More direct team communication and less documentation is a hallmark of agile methodologies compared to more traditional methodologies [?]. It's clear that face to face communication is more effective than the alternatives; scrum capitalizes on inter-team communication and potentially departmental collaboration [?].

3 Iterative Cycles

Another challenge presented by software development projects is the long development duration, typically ranging from one year to many. Projects have inherent scheduling and budget constraints and planning, scheduling and managing project work without a complete working domain knowledge can lead to wicked problems. This characteristic is actually a critical philosophy of waterfall methodologies. If a project is sequential, and almost predictive, then older software methodologies work fantastically. The straight forward sequence of phases greatly simplify a project and allow for planning and project status tracking. This approach falls short when business needs change midway through the project, which is inevitable if the planning and implementation take several years.

Agile methodologies approach to this problem is to separate development into time slices called iterations. An iteration can range in duration from two weeks to a month, and are defined by a series of user stories or features. The development is carefully planned and calculated according to how long a developer thinks it will take, and they use that information to put an iteration together.

3.1 Iterations

Iterations are basically incremental movements forward for a project. Each increment takes the form of a depth first movement rather than breadth first; the iteration is meant to develop a feature or user story, and not further every aspect of the software at the same time. This breaks the project up into easier to manage parts, and ensures there is always a working version of the software for the client to approve of [**?**].

Iterations are relatively short in duration compared to a fast-paced business. The development team can maintain quick and adaptive reaction time to ever changing requirements by changing what they have in their next iteration, or even midway through. This keeps development flexible. An example of an agile methodology that specifically excels at this problem is Extreme Programming. Developers using this methodology should use common sense and best practices to the extreme to plan and develop a project with vague requirements. The overall architecture is suppose to be created after the first iteration, and then the rest of the project will be implemented in a production environment to quickly determine what must be added or removed.

4 Evolutionary Development

A major challenge of software engineering is the ability to change a system without causing problems to ripple throughout. The addition of a feature in the software could impact the rest of the system, and cause unforeseen problems. Traditional methodologies attempt to predict these problems and handle them beforehand, or not even deal with them because the software has been planned out from the start.

A core philosophy of Agile is to always have working software. This usually occurs in generations, or rolling releases of the software. Each patch, or generation, comes with additional functionally while maintaining previous stability. This can be difficult when a business is watching the development, as is standard with agile methodologies, but the common motivations for each party (high revenue in this case) provide a good common ground [**?**].

4.1 Test Driven Development

Test driven development is a widely adopted method of development, and is the accepted initiate action for agile methodologies. Creating a test not only focuses the developer on a specific aspect of the problem, but it is proven to aid in scientific programs implementation in an agile methodology context [**?**].

Each generation of the program has to be able to be as functionally effective as its previous generation and then also complete a new user story. Test driven development forces the developer to pass previous tests as well as the new ones, ensuring that the program doesnt break with the new addition. Testing easily quantifies the status of the project by showing what the program can do at any given time, up to the point of the tests.

5 Conclusion

The motivation to develop software has never changed. Developing good, stable and dependable software is ultimately the end goal for any project, and a solid process and methodology can have a huge impact on achieving that goal. Agile methodology shouldnt be thought of to be superior to traditional methodologies; it is simply different. It was thought of and constructed to solve a variety of different problems, including project communication, rapid response to variable requirements, and long development durations. Todays software needs to be more interactive and thus is almost impossible to design with every determinable use case in mind. This puts sequential and predictive methodologies at a disadvantage.

References

[1] Lan Cao, Balasubramaniam Ramesh, and Tarek Abdel-Hamid. Modeling dynamics in agile software development. *ACM Trans. Manage. Inf. Syst.*, 1(1):5:1–5:26, December 2010.

[2] André Janus. Towards a common agile software development model (asdm). *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, July 2012.

[3] Oualid Ktata and Ghislain Lévesque. Agile development: Issues and avenues requiring a substantial enhancement of the business perspective in large projects. In *Proceedings of the 2Nd Canadian Conference on Computer Science and Software Engineering*, C3S2E '09, pages 59–66, New York, NY, USA, 2009. ACM.

[4] Magnus Thorstein Sletholt, Jo Hannay, Dietmar Pfahl, Hans Christian Benestad, and Hans Petter Langtangen. A literature review of agile practices and their effects in scientific software development. In *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, SECSE '11, pages 1–9, New York, NY, USA, 2011. ACM.

[5] Christoph Johann Stettina and Werner Heijstek. Necessary and neglected?: An empirical study of internal documentation in agile software development teams. In *Proceedings of the 29th ACM International Conference on Design of Communication*, SIGDOC '11, pages 159–166, New York, NY, USA, 2011. ACM.