An Explication of Popular Agile Software Development Methodologies

In its history, agile software development has witnessed a transition in the level of process specificity, area of emphasis from among the process steps, and means of communication through which new features can achieve customer objectives in an adaptive, expedient manner. This paper presents the history, applicability, and tradeoffs among several from among these popular methods by proceeding in order of: Feature-Driven Development (FDD), Scrum, and Extreme Programming(XP) as an attempt to transition from the least rigorously defined approach to the more heavily documented and specific approach. Schuh prescribes a classification scheme that is based on the methodology's approach to: management and programming activity, means for fostering a culture of communication, collaboration, and culture; concept of an iterative project flow; and tolerance for diverse project environments [1]. Independent of the methodology compared, there will be some jargon that is necessary to communicate ones understanding of the concepts native to the methodology, so certain keywords mapped common language constructs are also provided.

Feature Driven Development

Within the realm of FDD, systems enhancements and functionality are implemented based upon the functionality of components, referred to as *features*. To determine whether a system enhancement qualifies as a feature, a determination of the amount of time in which it can be implemented must affirm that no more than two weeks are needed during for development. As frequent deadlines may increase predictability and efficiency in a software development project [5], FDD uses two week timeframes for the team for each feature, but in terms of the iteration length in any given development cycle, FDD also provides guidelines for other phases of the iterative cyle. FDD approximates the ideal of agile development by prescribing that the majority of developer time is focused on meeting higher rates of requirements change and which results in lower complexity and risk [7], better feedback (loops), and higher success rates.

There are five phases to the FDD approach: develop an overall model; build a features list; plan by feature; design by feature (DBF), and build by feature (BBF), of which the latter two are iterative and are likely to consume roughly 77% of the time, as Coad and DeLuca estimate [7], using a FDD methodology. At the outset, an FDD approach is intended to ensure customer communications, accountability, and a plan by which enhancements can be built to contribute to the overall system result. In the first stage, a class diagram is one of the deliverable artifacts that sheds insight into the system, domain areas, and architectural connections existing between the classes. Likely to further encourage inclusivity of all programmers during the design phase, Ramsin and Paige [] argue that during the first stage during model building the Chief Architect may break the modelling team into sub-teams of no more than 2-3 to obtain diagram submissions from varying sources. The customer has opportunity to provide discretionary input as to overal progress at any point during development; however, provided there are no objections, a tenth of the way through the alotted timeframe for development, the "building a features list" phase begins. It is during this phase that customer input as to the developmental effort plan will be given, as it is when the conversion of methods from the class diagrams' modelling will be transformed to features, using a hierachical classification scheme – distinctive of FDD – will occur and features and feature-sets will be prioritized. If there exist multiple levels of managerial roles assigned in an FDD project, during the subsequent "plan by features" stage, which takes a very minimal amount of the project time, major feature sets and features may be assigned to chief programmers; while, in general, class ownership affords the accountability factor in this methodology with individual classes being owned by (non-chief) programmers.

Next, the iterative phases take the reigns, with "designing by feature" occurring first where the chief programmer will assign classes and the feature team(s) will create detailed sequence diagrams, to a level of detail such that data types and parameter types for the methods used in the classes are portrayed. The run-time interaction of objects ,which enables implementation sufficient to achieve the design goals, is

also included. Those features having been identified as requiring first-attention during the prioritization of features phase, previously, will be developed by the features team in this phase, with assistance provided by members of the effort aware of vital technical or architectural details having an impact towards the feature being developed in the current iteration.

The needed classes having been crafted by the programmers to which the chief programmer assigned them, the effort next shifts into the "build by features" phase, where the created classes must be verified as conformant to design standards internal to the developing company or as the customer requires. With functional and unit testing of the developers' code completed, the classes can be promoted to the current build, at which point a responsibility shift upon the chief programmer is due to ensure the classes required to implement the feature are all included in the build.

FDD vs. Scrum

While FDD places an emphasis on design, at the outset, Scrum is a management-centric approach that allows for a more direct line-of-sight into the level of productivity for programmers, eliminating any possibility for occlusion by personnel or other managerial resources. Further, where FDD provides accountability by way of class ownership, Scrum places more responsibility on the shoulders of the team adhering to this methodology than may be so when compared to other agile methodologies [1]. It is the role of an individual designated as the *Scrum Master* to ensure the project team and customer groups both conform to the principles of Scrum, but perhaps more precariously, it is the job of the product owner to prioritize the product backlog and to tell the project team what they will be developing at the beginning of each sprint. The approach to management in terms of human resources allocated per each methodology are clear in that Scrum declares 4 roles: a self-managed team, scrum master, product owner, and everyone else; yet, FDD depends upon six key roles: project manager, chief architect, development manager, chief programmers, class owners, and domain experts.

With FDD there are 5 key processes, with a multitude of guiding descriptors provided towards each, whereas with Scrum, there are 3 key processes and, by comparison, little guidance in terms of formal descriptors.

There have been cases where FDD has been shown to have varying degrees of flexibility in terms of the iterative-phasing of the approach vs. the planned portions of the approach, whereas Scrum is fairly standard and issues a single piece of guidance regarding the period of time in which development iterations should last. Scrum is explicit in stating that development iterations, known as *Sprints* should last no more than 30 days and that the latter two phases of Scrum may be iterated over; with FDD, as well, it is most often the case that two phases - 4 and 5 - are iterated over, but occasionally the entire 5 phases will be iterated, instead.

The means for fostering communication and culture for each method are somewhat understandable based on the roles for accountability and realizing Scrum further places accountability on its own shoulders by ensuring once a Sprint has begun the project team has discretion and clearance to make such decisions as necessary to complete its tasks. FDD places more of an emphasis on upfront design and may be better suited for instances where managerial oversight is not as much a concern as utilizing the full capabilities of each programmer – who has ownership of specific classes – by eliciting their involvement in more of the design aspect of the development process. To differentiate the process flow and structure of FDD and Scrum further, Scrum prescribes a regimen of daily meetings to close communication loops which could arise among the team and customers and for circumventing any roadblocks on a routine basis.

FDD vs. Extreme Programming (XP)

XP is perhaps the most stringently defined methodology in terms of the developmental processes it demands from among all agile methodologies, and it is unique in that it also may be the most pressureinducing

while -rewarding approach for development teams. XP is unlike FDD or Scrum in that it places little to no focus on the managerial side of software development; rather, all focus is on programming. Programmers, likewise, are responsible for their own analysis and management. XP development is completed by programmers from start to end, is declarative of 12 best practices – versus 8 practices for FDD, and mandates strict testing procedures be implemented. With XP, it is declared exactly how testing is to fit into the development process, even.

Concerning the timeframe in which XP projects are completed and iterations will elapse, an XP project may be the shortest of all agile project types, as with XP, teams will deliver a fully programmed, production-worthy version of the entire system in 1-4 weeks – compared with using an FDD approach which might require up to 2 weeks just to develop a single feature.

Conclusion

With all agile development methodologies, the approach can be understood to emphasize: communication, feedback loops, and that the overall process will result in a lesser volume of documentation than with plan-based approaches to development. FDD, Scrum, and XP are likely to succeed in these areas. An FDD approach emphasizes an atmosphere of responsiveness, courage, and respect; Scrum emphasizes transparency; and XP emphasizes functional, tested, responsive code without any non-programmers staffed to the project team.

References:

1. Schuh, Peter. "Integrating agile development in the real world",1st Ed. Hingham, Mass. :Charles River Media, c2005. pg.18 (Header: "Methodology Descriptions")

2. http://www.agilemanifesto.org/iso/en/principles.html

3. R. Ramsin, R. Paige."Process-centered review of object oriented software development methodologies": ACM Computing Surveys (CSUR), Volume 40 Issue 1, February 2008.

4. http://xprogramming.com/what-is-extreme-programming/

5. T. Potok and M. Vouk, "The Effects of the Business Model on the Object-Oriented Software Development Productivity," *IBM Systems Journal*, vol. 36, no. 1, pp. 140-161, 1997.

6. http://nebulon.com/articles/fdd/download/fddprocessesUSLetter.pdf

7. http://web.archive.org/web/20040426030247/http://www.pcoad.com/download/bookpdfs/jmcuch06.pdf

8. C. Larman, *Agile and Iterative Development: A Manager's Guide*. Boston: Addison Wesley, 2004.

9. https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf#zoom=100

Plan-driven vs. Agile Methods

Following the era when programmers could get by using ad-hoc team formations or exist as individual subject matter experts, a need for structured, process-driven approaches to software development arose; hence the formation of plan-driven method. Against the test of time, the waterfall method has endured and been a topic of study for many research activities, from those noting cases where the waterfall method has been a success to catastrophic illustrations of its inability to produce a less than quality outcome. [1] presents an empirical study comparing the various plan-driven methods, such as the waterfall method, to many agile methods, using findings discovered in industrial and academic settings.

While the pros and cons for the two methods are presented, the former as compared to the latter method offers few real advantages, so only the section addressing agile methods is divided into sections covering its advantages and disadvantages in [1]. Generally, plan-driven methods were a step forward for software engineering (SWE) in terms of structure due to their declaration and insistence upon phases such as: requirements analysis, design, coding and implementation, testing, and maintenance. While, the plan-driven approach began to be viewed as less relevant given the emergence of more customer and results-oriented approaches in the realm of agile. Disadvantages cited for the plan-driven approach include: the architecture and design specification has to be complete before implementation begins – ideas that bring business value can not be conceived along the course of development; programming work is only concentrated in the programming phase – it is not iterative; testing is done in the end of the project – so any broken aspects can have far-reaching effects when determined at a late phase of development; quality assurance is handled in a formal way – while consistency checks are performed, the quality that is ensured may emphasize importance on matters of lesser significance as compared to the approach agile QA takes.

A recent article [2] cites that a programmers lament may be that "to capitalize on a new idea, as soon as it is discovered, you have only of few weeks to implement it; then you have to start all over." An unmistakable disadvantage to agile-development is that there is a lack of knowledge of requirements upfront sufficient to allow modelling of the architecture for a proposed system. As a result, agile-developed systems do not scale well and may become utterly waste worthy, if the system has to be re-engineered due to architectural shortcomings. [1] cites testing as a major bottleneck in agile development due to its need to remain afford an integrated test environment, inclusive of different systems and architectures (though this may not be the case with some flavors of an agile approach such as those where co-location is mandatory). An additional disadvantage [1] cites is that customers have to remain committed throughout the duration of the agile development effort to reap its rewards, which places constraints on their availability and resources and shifts more risk onto them.

While an abundance of studies have been completed studying the pros and cons, comparing agile to plan-driven approaches, [1] clearly asserts that previous systematically executed studies have led to inadequate results-data, yet remains assertive that agile methods are more advantageous, by large, stating: "many issues commonly raised for the plan-driven approach are not raised for the incremental and agile approach." Customers typically feel that the work environment fostered by an agile approach is more work-life friendly and that communication and knowledge-transfer are enhanced, yet there may be an increased need for those who initially begin an agile effort to remain until the effort is complete due to lessened interchangeability of team members in an agile-effort context. Despite the advantages of knowing architectural requirements, mishaps can happen even with plan-driven approaches, so with the exception of the London Ambulance Service CAD System development effort, only in mission-critical instances are upfront investments warranted as is the case of a plan-driven approach.

References:

[1] Kai Petersen and Claes Wohlin. 2010. The effect of moving from a plan-driven to an incremental software development approach with agile practices. Empirical Softw. Engg. 15, 6 (December 2010), 654-693. http://dx.doi.org/10.1007/s10664-010-9136-6

[2] Jacob Loveless. 2013. Barbarians at the gateways. Commun. ACM 56, 10 (October 2013), 42-49. http://doi.acm.org/10.1145/2507771.2507779