Journal of Software Engineering and Applications, **2012, 5, 62-68**
doi:10.4236/jsea.2012.52010 Published Online February 2012 (http://www.SciRP.org/journal/jsea)

Laboratory Driven, Lean-to-Adaptive Prototyping in Parallel for Web Software Project Identification and Application Development in Health Science Research

Zachary Dwight, Alexa Barnes

Department of Pathology, University of Utah, Salt Lake City, USA.
Email: zach.dwight@path.utah.edu, alexabarnes@msn.com

Received December 21st, 2011; revised January 24th, 2012; accepted February 3rd, 2012.

ABSTRACT

Clinical research laboratories, bioinformatics core facilities, and health science organizations often rely on heavy planning based software development models to propose, build, and distribute software as a consumable product. Projects in non-agile software life cycles tend to have rigid "plan-design-build" milestones, increasing the amount of time needed for software development completion. Though the classic software development approach is needed for large-scale and organizational projects, clinical research laboratories can expedite software development while maintaining quality by using lean prototyping as a condition of project advancement to a committed adaptive software development cycle. Software projects benefit from an agile methodology due to the active and changing requirements often guided by experimental data driven models. We describe a lean to adaptive method used in parallel with laboratory bench work to develop quality software quickly that meets the requirements of a fast-paced research environment and reducing time to production, providing immediate value to the end user, and limiting unnecessary development practices in favor of results.

Keywords: Agile Software Development; Bioinformatics; Lean; Prototyping; Adaptive

1. Introduction

Clinical research laboratories often use rapid application development and agile software methods [1,2] for computationally intensive tasks related to health science and biological data analysis, collection, modeling and simulation. These tasks, however, are often done *ad hoc* and the development process is selected in a similar manner, increasing the amount of time needed to complete the project, limiting user feedback, and producing poor documentation. Laboratories, often with funding and/or publishing deadlines, are inefficient with the planning, development, and documentation of chosen software projects in an attempt to develop software due to internal and external constraints. Expertise is often lost in transition between the laboratory end users and the software development team and many research laboratories prefer to develop in-house using available programming skill sets, which may not include the best technology for the computational task.

Though many health science researchers are unfamiliar with the best practices of agile software development, previous work suggest a core set of practices does exist that can be adopted by similar biomedical, bioinformatics, and health science professionals to produce quality software in a reasonable time frame [2-6]. From this previous work, our software development team has implemented an agile methodology that has been successful [7] in creating, distributing, and improving internal and external software projects.

2. Methods

The agile method in use in our research environment aids in the development of quality software in parallel with health science research. Lean-to-adaptive prototyping in parallel (L2APP) focuses software efforts on results rather than features or tasks. Quality experiments produce quality results and quality software produces quality analyses, predictions, simuations, and modelling. When applied to health science research, L2APP is successful due to its flexibility to change, conditional lean phase for project commitment, emphasis on results, and delivery of value in parallel.

2.1. Discovery

With each experiment performed, laboratory staff and researchers should track and identify redundant tasks, pro-

cesses, analyses, calculations and novel methods/and or results that could benefit from software or be transitioned into a system of software services. The ideas for software projects are then presented or pitched to the software team. Though a high percentage of software will be laboratory driven, it is not uncommon for software teams to observe opportunities for projects, which could also be considered with consultation with the laboratory staff and researchers.

2.2. Lean Prototyping

Lean prototyping or lean software development is an attempt at developing quality software of value with the least amount of effort [8]. Software teams following lean guidelines are exempt from producing documentation and processes that do not add value to the current project. Developers are asked to focus on quality, value, and reusability and deliver only the software product. The lean ideal lends itself well to web development projects that are less computationally intensive than many desktop applications due to processing constraints.

2.3. Adaptive Software Development (ASD)

Though a variety of agile software methodologies exists that reduce time to project completion, the adaptive software development model is organized in a three—phase cycle that can best be applied to our laboratory driven requirements, results, and analyses. In this model, as seen in **Figure 1**, the three primary phases of adaptive software development are speculation, collaboration, and learning [1]. Speculation is the ability of the development team and end users' to formulate a development direction for each iteration through the software process. Applied to research laboratories and software teams, speculation is the agreement between a laboratory member (end user) and the software development team describing the general, but not complete, idea for a prototype or prototype iteration. Once this agreement or speculative idea has been presented, the software team can continue developing the prototype. The next phase in the cycle is collaboration. Collaboration is the cooperation of the end user and the software team. Change, requirements, and limitations can evolve from either laboratory results or computational setbacks. Collaboration between the two parties, laboratory and software, can ease and/or overcome limitations, better adapt to change, and better define requirements. The third phase in the cycle is learning. One of the strengths of the adaptive development model is the learning phase. The learning phase allows both software and laboratory teams to identify and acknowledge issues, changing requirements, and research opportunities that were discovered in the previous cycle of both laboratory work and/or software development cycle. This three-phase development environment places emphasis on quality benchmarking results between the software and laboratory rather than tasks, features, or scheduled development milestones.

3. Lean-to-Adaptive Model

A frequent complaint of agile software development is the lack of a structured and concrete project timeline. In this effort, our model at the least contains a definable beginning and a reasonable end point. A laboratory need or opportunity is the focus for both the beginning and the end of a software project. A need arises in a research environment and once that need has been fulfilled with developed software, the software project is considered complete. Often, smaller software projects are fulfilled simply by the development of the lean prototype, never entering the adaptive phase. However, research environments are constantly changing software requirements based on experimental outcomes especially in the case of computational and simulation model development. To account for both smaller, tasks oriented projects and model based research requirements, the lean-to-adaptive model relies on the following phases: need/opportunity identification, lean prototype development, lean evaluation, and the adaptive software development cycle. The adaptive software development life cycle follows an agile methodology consisting of development, production, quality assurance, and experimental assurance within the larger context of previously mentioned phases of speculation, collaboration, and learning. **Figure 1** displays overall flow of development of model.

3.1. Opportunity Identification

Laboratory work and clinical environments are burdened with repetitive tasks, much of which can be automated to allow laboratory staff and researchers more time to evaluate quality results and help reduce processes prone to error. The opportunity identification phase is the communication to the software developer(s) of a need, opportunity, or requirement of the laboratory that could be aided by either automated or user driven software.

3.2. Lean Prototype Development

A lean prototype is developed in an easily accessed form,

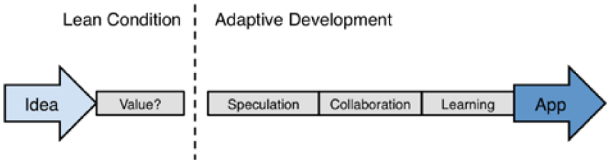

Figure 1. Flow chart of development for lean-to-adaptive prototyping in parallel model.

such as a web application or script according to the following standards and conditions:

- Lean development efforts focus on value to the customer. The lean prototype only has value if it fulfills or partially fulfills the requirement or need identified by the research group.
- Ignore documentation, organization, and performance. The value stream, or the effort needed to develop a prototype of value, requires only that a prototype be delivered.
- In lean software development, no individual upstream (the software group) can build and/or code without a downstream request (research staff). The end users should champion (also known as "pull") their own requests, not the software team.
- Use local resources and expertise to build the prototype. The technology requiring the least amount of effort to initiate, develop, and distribute should be used.
- Developers should save/store all reusable code for future prototypes and/or projects. Reusable code is valuable for the current prototype and possibly for future prototypes/projects.
- Distribute the prototype in the most accessible form available. End users should not have to compile source code or run complicated install procedures.

Once the prototype is complete (can demonstrate or partially demonstrate value), a walkthrough is performed with the end user or research group for evaluation.

3.3. Evaluation of Lean Prototype

The evaluation of the lean prototype is an important step in this model. The evaluation has three possible outcomes as decided by the end user(s). The first outcome is commitment. If the lean prototype presents enough value to the research group that it could be immediately used, the lean software project moves into the adaptive development phase in collaboration with the laboratory staff. The second outcome is a renewal. Renewal indicates the prototype has value but not enough to commit to as is. Value that is missing should be communicated to the software team and the prototype renews its lean development cycle. The last possible outcome is suspension. If the prototype has little value the prototype is suspended and archived appropriately. Many factors can reduce value of a lean prototype including changing requirements, external resources, commercial software, and poor timing.

3.4. Adaptive Software Development

Once a prototype has the commitment of both the end user and software team, the prototype enters a larger, iterative development cycle referred to as adaptive software development (ASD). The three guiding principles or phases of ASD are speculation, collaboration, and learning as described previously [1]. The three phases divide the components of the software development life cycle appropriately while being descriptive enough to remind developers and customers alike of the goal of the phase and the cycles contained. **Figure 2** details the steps of development and the parent phases.

3.4.1. Speculate

Speculation describes the initial or repeated identification of software goals and direction during the development process.

3.4.1.1. Prototype Commitment

Prototype commitment is the *de facto* beginning of a software project. By commiting to the prototype, the adaptive software development life cycle begins and the lean prototype phase ends. Without commitment from both end users and software team, the project should not begin.

3.4.1.2. Laboratory Cycle Plan

A whiteboarding session, presentation, quick conversation or document can all be used to identify the goal of the current software development cycle. In the laboratory-driven model however, a continued lean approach to requirements gathering defines the current cycle objectives. The goal of all development iterations is to add value and fulfill the laboratory requirement and/or need. A list of features with their resulting value descriptions substitute for mockups, documentation, and software object modelling. The software team does not need to know what the feature consists of, but rather what result the feature will deliver. Results based development allows software professionals more creativity in how they choose to acquire a result. Once the software team has their feature list, software development begins.

3.4.2. Collaborate/Development

Collaboration during the development life cycle involves the parallel development of software and experimental results. The software being developed, in its current form, may be used as soon as it has value to the laboratory staff and researchers. Reusable code stored in a repository could also quicken the current development cycle. Developers should also be aware of code being written that has possible future use and archive it accordingly [3,9]. Collaboration also promotes the continued discussion of requi-

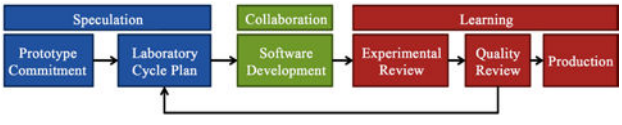

Figure 2. Adaptive software development model.

rements and results as both are being developed. The laboratory and software staff communicate any advances, changes, limitations, and needs during this cycle so that either may react accordingly and adjust the cycle plan or experimental design. Collaboration and knowledge-share allow for developers to ignore dated practices such as UML and technical specification documentation [9]. If software developers and researchers understand their requirements, results, and science well enough, creating "middleman" documents is unnecessary and considered non-agile.

3.4.3. Learn

The learning phase consits of software validation and assurance of both quality and experimental results. If the software proves to be useful and meet all current requirements (has substantial value), the software project becomes a software product and is moved into production in an easily installed form or via web application or service.

3.4.3.1. Experimental Assurance

Experimental results are used to validate the software. Predictive computational modelling, simulations, and design tools should produce results that are reproducable, valid, and applicable in a laboratory environment. Data analysis software and calculators should reproduce quantitative values calculated by hand, automated testing, or ad hoc in statistical software such as Microsoft Excel or STATA. Workflow management software should ensure data provenance, delivery, and accuracy as it moves through a system. If the software does not produce, simulate, or deliver quality data, predictions, or laboratory aid than the software must reenter the development cycle. One caveat to this review is poor experimental data quality. Models are only as good as the data they are built upon. In this review, both the software team and the laboratory staff should identify possible issues stemming from either effort and the review should involve validation of not only the software but of the experimental results.

3.4.3.2. Quality Assurance

Quality assurance is the process of cleaning code, debugging, and ensuring efficiency and performance for the software being developed. Review of the software in context of computational best practices can often identify mistakes that lead to a failure of the experimental assurance review. If a bug has caused a calculation error, the software developer should rerun the experimental benchmark before returning the software to the beginning of the development cycle.

Using automated testing and "test as written" approach is highly recommended and can save valuable time [3], yet manual testing done by the end user can be more valuable in instances of GUI driven calculations. Test cases make assumptions about the end users' actions which works well for command-line style processes, but is limited for event-driven applications or rich web applications requiring real-time interaction.

3.4.3.3. Production

Once the software and experimental results agree or the software is useful for laboratory consumption, the software is moved into production. In an adaptive environment, it is easier to use web-based applications or services for the production environment. Any updates to the software are made in a single server location and users would not have to reinstall or possibly fall behind the current software version, patches, fixes, or updates as commonly seen in desktop installations.

4. Model Assessment

The L2APP model is an agile software development model focused on providing quality software quickly by working in parallel with a laboratory or research group. Working in parallel can lead to knowledge transfer based solely on proximity that reinforces the project goal and helps both groups understand the context of their work [10]. This parallelism promotes one of the core phases of adaptive software development: learning.

4.1. Focus on Results, Not Plans

A solid software plan in a classic SDLC can solidify a project and provide all involved a shared vision for the project. Unfortunately, changes to the plan can have disastrous effects on the project as a whole. In our experience using this model, experimental results continually change or update software requirements. In a research environment, software development must be driven by the end result and not by overly embellished planning. With each iteration, the collaborative nature of the software development cycle and the learning focus of the reviews put forth a model that promotes improvement of the results through iteration rather than a heavier initial planning phase. Focusing on the result, rather than planning the path to acquire the result, saves time and effort.

4.2. Time to Production

Using this model, time to production for software ultimately depends on quality experimental and software practices working in parallel. By working in parallel, setbacks in the laboratory can also setback software development. However, being an agile software methodology, those setbacks, changes, failures have a small impact on the overall time to completion for a software project. An error made in the laboratory is communicated and known

to the software team and the software can re-enter another iteration of development, thus protecting the overall project timeline. Using other methods of software engineering, the software team would wait until the experimental results were gathered, analyzed and validated before even planning a software product. The L2APP model rewards the quality development of experiments and software in parallel by reducing the time to production. When the experimental data is gathered and validated, the software should be ready to move into production.

4.3. Value during Development

Since a project cannot enter the adaptive development life cycle until it has value to the laboratory or research group, the software can be used immediately upon commitment. Constant use of the software, in any form, is valuable to both the researchers and the software team. The researcher or laboratory staff member can continuall validate the software with each iteration and the software team receives valuable feedback. The L2APP model is a real-time trade off of value: the software team provides tools to aid the researcher and the researcher provides feedback/validation.

4.4. Projects and Portfolio Have Value

Any project in this model begins as a lean prototype and is immediately discarded if it lacks or loses value. By using a conditional lean prototyping phase, the model introduces its own project portfolio quality control. Software teams rarely work on a single project at a time and often have a portfolio of current projects. It is important to only add projects that add value to the portfolio just as it is important to add only features to software that have value. The conditional lean prototyping phase allows the software and research groups to protect the project portfolio from losing value and wasting resources on software that has little value. Using this model, the software team's project portfolio is protected from poor project selection.

4.5. Lean Prototypes Are Often Enough

An interesting outcome of this model in some instances is the acceptance of the lean prototype as a production worthy application. End users have a difficult time realizing and identifying their needs, thus the choice of agile methods of software development. With each iteration, the end user and software team work together to better define the needs and eventually build a solid software product. However, we've seen cases where end users find enough value in the prototype and its placed into production for use. At first the end user may have had grandiose requirements, but quickly realized a prototype containing partial requirements was all that was truly needed.

4.6. Lean Methods Continue Throughout

A side effect of the initial lean prototyping phase that we've experienced is continuation of lean ideas and goals throughout the adaptive development as well. Though there is a clear distinction made between the lean prototyping phase and the adaptive development phase, we've seen smooth transition between the two, as the lean methodology fits well with the adaptive methodology. Software teams often have difficulty switching between software development methods such as SCRUM, classic SDLC, and XP because the relationships of stakeholders, coding standards, and development organization are quite diverse [11]. The L2APP model promotes a transition between a very lean prototype and what could be considered a less lean (adaptive) prototype.

4.7. People

The individuals involved are very important to the success of a project and application of a software methodology [2,10]. Some individuals do not conform well to an agile software environment and any methodology used would have to be adjusted to accommodate. It is possible to educate individuals on the best practices of agile development [12] but education and the willingness to adopt do not go hand-in-hand. Adoption of an agile method is easiest for those who are lean by nature and difficult for those who thrive on planning, meetings, and paperwork. This is also true for the method presented in this publication.

4.8. Speed vs Results

Experimental results are often slow to acquire, organize, and analyze in hopes of ensuring the upmost quality and reproduciability. Web development however has been seen to focus merely on speed [9]. In our experience using an agile method for web software in a research environment, timing can be an issue as quality is non-negotiable in experimental design but seen as negotiable in prototyping.

5. Discussion

Previous research consolidated the characteristics of agile software projects as incremental, cooperative, straightforward, and adaptive [1]. The L2APP model shares these characteristics with other agile methodologies by promoting cyclical development, collaboration between research and software during development, being easy to understand and modify, and flexible to changing requirements without disastrous effects to the project or timeline. It is not recommended however that our interpreta-

tion of an agile methodology is better than those with less or more definition, but rather an interpretation that we find useful in our research environment.

Our model modifies the adaptive development software model by introducing parallelism, lean principles and a conditional lean prototyping step. Parallelism can reduce resource slack, improve collaboration, increase stakeholder understanding of project context, and constantly align the timelines of the experimental and software groups. Parallelism is however dependent upon quality in this model. Quality experimental results can define requirements for software development or be used to assess the software product. Quality software can be used to aid in experimental design, prediction and visualization. In either case, a lack of quality could continually return the software project to the development cycle.

Lean principles are introduced to reduce time to production. By not providing documentation, mockups, or object models during development, the software team can focus on the result and research requirements. In health science research, assays are only as good as their outcome and the same is true for software projects. In contrast, some very important details are left out of the development cycle including quality objectives, risk management, and detailed specifications of software [9].

A concern for many bioinformatics teams is the ability to manage and funnel projects into the development environment. Academia promotes collaboration and this is also true in decision-making. As more groups, researchers, and staff become involved in a project, the ability to manage the portfolio of projects is crucial in prioritizing resources. The initial lean prototype condition introduced in this model helps preserve the project portfolio's value. Most projects received by our team have value in their intent, yet a small percentage quickly lose value due to changing requirements, limitations of technology, and the discovery of external software. The lean prototype provides validation of the proposed software project by 1) ensuring technological feasibility and 2) asserting project value.

Bioinformatics groups and research laboratories have a large reference set in which to learn software methods that best fit their expertise, talent, motivation, and managent style. Unfortunately, very few of the references are creble, rigorous or detailed enough for academic acceptance [11,13-15] or adoption and only recently has software delopment been a focus for bioinformatics environments [3].

Interesting future work would include a survey of not only agile software practices in bioinformatics, but adoption success, project management techniques [16], and organizational optimization for successful research driven software projects. In this publication, we've described an agile method of software development that fits well with smaller, research driven environments lacking in dedicated project management resources in hopes of spurring research into the best practices of bioinformatics, health sciences, and laboratory driven software projects.

6. Conclusion

The lean-to-adaptive prototyping in parallel method is a useful agile software methodology that can be used by teams in research environments to build quality software quickly and manage the value of the projects committed to.

7. Acknowledgements

We would like to thank the Wittwer DNA Lab at the University of Utah, Canon US Life Sciences Inc., and Idaho Technology Inc. for their consistent feedback in the development of internal and public software, which led to the creation of the L2APP model.

REFERENCES

[1] J. Highsmith, A. Cockburn and B. Boehm, "Agile Software Development: The Business of Innovation," *Computer*, Vol. 34, No. 9, 2001, p. 3. doi:10.1109/2.947100

[2] D. W. Kane, M. M. Hohman, E. G. Cerami, *et al*., "Agile Methods in Biomedical Software Development: A Multi-Site Experience Report," *BMC Bioinformatics*, Vol. 7, 2006, p. 273. doi:10.1186/1471-2105-7-273

[3] K. Rother, W. Potrzebowski, T. Puton, *et al.*, "A Toolbox for Developing Bioinformatics Software," *Brief Bioinform*, 29 July 2011.

[4] J. Pitt-Francis, M. O. Bernabeu, J. Cooper, *et al.*, "Chaste: Using Agile Programming Techniques to Develop Computational Biology Software," *Philosophical Transactions of the Royal Society A*, Vol. 366, No. 1878, 2008, pp. 3111-3136.

[5] R. S. Sadasivam, K. Delaughter, K. Crenshaw, *et al.*, "Development of an Interactive, Web-Delivered System to Increase Provider-Patient Engagement in Smoking Cessation," *Journal of Medical Internet Research*, Vol. 13, No. 4, 2011, p. e87. doi:10.2196/jmir.1721

[6] K. Gary, A. Enquobahrie, L. Ibanez, *et al*., "Agile Methods for Open Source Safety-Critical Software," *Software*: *Practice and Experience*, Vol. 41, No. 9, 2011 pp. 945- 962. doi:10.1002/spe.1075

[7] Z. Dwight, R. Palais and C. T. Wittwer, "uMELT: Prediction of High-Resolution Melting Curves and Dynamic Melting Profiles of PCR Products in a Rich Web Application," *Bioinformatics*, Vol. 27, No. 7, 2011, pp. 1019-1020. doi:10.1093/bioinformatics/btr065

[8] S. Raman, "Lean Software Development: Is It Feasible?" *Proceedings of the* 17*th Digital Avionics Systems Conference*, Vol. 1, 1998, pp. C13/1-C13/8.

[9] R. Baskerville, B. Ramesh, L. Levina, *et al.*, "Is Internet-Speed Software Development Different?" *IEEE Software*, Vol. 20, No. 6, 2003, pp. 70-77. doi:10.1109/MS.2003.1241369

[10] A. Cockburn, J. Highsmith and B. Boehm, "Agile Software Development: The People Factor," *Computer*, Vol. 34, No. 11, 2001, pp. 131-133. doi:10.1109/2.963450

[11] J. Erickson, K. Lyytinen and K. Siau. "Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research," *Journal of Database Management*, Vol. 16, No. 4, 2005, pp. 88-100. doi:10.4018/jdm.2005100105

[12] V. Devedzic, S. Milenkovic, *et al.*, "Teaching Agile Software Development: A Case Study," *IEEE Transactions on Education*, Vol. 54, No. 2, 2011, pp. 273-278. doi:10.1109/TE.2010.2052104

[13] T. Dyba and T. Dingsøyr, "Empirical Studies of Agile Software Development: A Systematic Review," *Information and Software Technology*, Vol. 50, No. 9-10, 2008, pp. 833-859. doi:10.1016/j.infsof.2008.01.006

[14] T. Dyba and T. Dingsoyr, "What Do We Know about Agile Software Development?" *IEEE Software*, Vol. 26, No. 5, 2009, pp. 6-9. doi:10.1109/MS.2009.145

[15] M. Aoyama, "Web-Based Agile Software Development," *IEEE Software*, Vol. 15, No. 6, 1998, pp. 56-65. doi:10.1109/52.730844

[16] D. Karlstrom and P. Runeson, "Combining Agile Methods with Stage-Gate Project Management," *IEEE Software*, Vol. 22, No. 3, 2005, pp. 43-49. doi:10.1109/MS.2005.59

