A Case Study: Agile Development in the Community Laser-Induced Incandescence Modeling Environment (CLiiME)

Aziz Nanthaamornphong
Department of Computer Science
University of Alabama
Tuscaloosa, Alabama 35487, USA
ananthaamornphong@crimson.ua.edu

Karla Morris, Damian W. I. Rouson,
Hope A. Michelsen
Sandia National Laboratories
7011 East Avenue
Livermore, California 94550-9610, USA
knmorri@sandia.gov, rouson@sandia.gov,
hamiche@sandia.gov

***Abstract*—The multidisciplinary requirements of current computational modeling problems preclude the development of scientific software that is maintained and used by selected scientists. The multidisciplinary nature of these efforts requires the development of large scale software projects established with a wide developer and user base in mind. This article describes some of the software-engineering practices adopted in a scientific-software application for a laser-induced incandescence community model. The project uses an Agile and Test-Driven Development approach to implement the infrastructure for the development of a collaborative model that is to be extended, modified, and used by different researchers. We discuss some of the software-engineering practices that can be exploited through the life of a project, starting with its inception when only a hand full of developers are contributing to the project and the mechanism we have put in place in order to allow the natural expansion of the model.**

***Index Terms*—agile development method, test driven development, computational software development, software engineering**

I. Introduction

Over the years, the Agile method has become a powerful approach in the software-development process. The Agile method focuses on incremental and iterative development where requirements, specifications, design, implementation, and testing continue throughout the project lifecycle. It was recognized as the new methodology used to solve some of the problems in traditional software-development processes, such as the Waterfall model [9]. One of the problems with the Waterfall model is that all user requirements must be specified at the beginning of the project. In practice, stakeholders can rarely identify all requirements in the initial stages of the project. Under the Waterfall model, requirements can be changed in the later stages of the software-development cycle, but this process can be extraordinarily costly. Agile methods promote an iterative mechanism for developing software, and they further increase the iterative nature of software lifecycle by introducing a *test-code-refactor* process in each iteration of the software-development cycle. By implementing a number of software-engineering practices, Agile ideas can be adopted to reduce the cost associated with changing requirements. Rather than focusing a lot of effort on the up-front design analysis, small increments of functional code are implemented according to customer needs. Test-Driven Development (TDD) is one of the methodologies in Agile that was introduced to minimize the efforts required from source-code developers.

TDD is a high software quality approach in which the tests, called unit tests, are written prior to the functional code itself. In particular, TDD helps software developers ensure that the software is built under the user requirements. It also helps developers ensure that the scopes of the software features are addressed. In fact, the TDD approach came with the introduction of eXtreme Programming (XP) [4]. XP is a discipline of software development based on the values of simplicity, communication, feedback, and courage [3]. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation. XP requires control at all levels: "project planning, personnel, architecture and design, verification and validation, and integration" [2]. The process of TDD is illustrated in Figure 1. First, the developers write the test cases for a particular functionality. Because of unwritten code they all fail. Next, the developers implement the functions based on the requirements to make the tests pass. Developers only refactor the code after all the tests pass successfully. During the refactoring process, the existing code is changed without changing its external behavior [7]. Consequently, TDD can be thought of as a programming technique that is based on a simple rule: "Only ever write code to fix a failing test" [11]. The TDD cycle is shorter than the traditional software-development cycle (*design-code-test-debug*) because it focuses on building exactly what is needed and uses immediate feedback to reduce regression errors.

In this study, we adopted the Agile and TDD methodologies to develop new scientific software. Subsequently, we investigated what benefits are gained by the scientific-software application as a result of the Agile and TDD approach. Here,

SE-CSE 2013, San Francisco, CA, USA

we describe why we decided to use the Agile methodologies on this project. In his work, Miller identifies nine main characteristics of the Agile method [14]; we map those nine characteristics to the characteristics of scientific software-development nature. Table I shows how each of Miller's defined characteristics of the Agile method support developers or scientists.

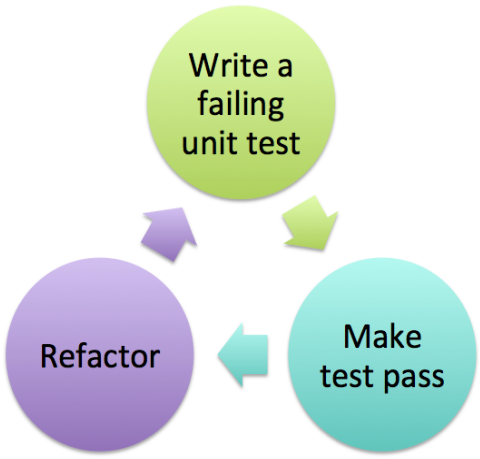

Fig. 1. Overall Process of Test-Driven Development

This paper reports results from a case study where a scientific-software application, implemented with object-oriented (OO) Fortran, was developed to facilitate the analysis of experimental data from a common soot-detection technique used in combustion diagnostics. The technique, laser-induced incandescence (LII), involves irradiating soot particles with intense laser light to heat the particles to a temperature high enough to emit measurable incandescence [17]. Particle temperatures range from 3000 K to 4450 K. At these temperatures, particles can anneal, sublime, and oxidize. They can also cool by radiation, thermionic emission, and conduction to the surrounding atmosphere. These mechanisms and how they respond to experimental conditions control the evolution of the LII signal. Thus, understanding the interplay between these mechanisms and experimental conditions is critically important for analyzing the measured signal. Numerous research groups have developed energy- and mass-balance models to describe particle heating by laser absorption, cooling by conduction, radiation, and sublimation, and mass loss by sublimation [13]. Some models additionally include heating by oxidation and annealing, cooling by thermionic emission, and mass loss by photodesorption and oxidation. The agreement among these models can be described as poor at best [13]. In an attempt to bring consensus to the understanding of the technique and consistency in the analysis of results from different research groups, we have established a community-based modeling environment designed to promote more collaborative model development. In order to initiate the project, we have selected an LII model that has been validated against a range of experimental data as our base case. This model solves the energy- and mass-balance equations to yield particle temperature and size and resulting LII signal as a function of time during and after the laser pulse. This model includes heating by laser absorption and oxidation and cooling by conduction, sublimation, and thermionic emission. Mass is lost via sublimation, photodesorption, and oxidation.

The remainder of this paper is organized as follows. Section 2 provides the description of the case study. Section 3 discusses the lessons learned from the case study along with the implications. Finally, the conclusions and future work are presented in Section 4.

TABLE I
THE CHARACTERISTICS OF AGILE DEVELOPMENT AND SCIENTIFIC-SOFTWARE DEVELOPMENT

Agile Development	**Scientific Software Development**
1. Modularity of development activities	Scientists and developers have certain activities in the software-development process. For example, performing the experiment, examining the result, and comparing the result.
2. Iterative with cycles	Scientists or developers might modify both the model and code when they received unexpected results. They repeatedly develop the model, perform the experiment, and compare the result in several iterations [16].
3. Time-bound with iterations	Since scientists need to run the experiment as quickly as possible, they work with limited time and a defined schedule in each iteration.
4. Parsimony	Scientists and developers do not perform unnecessary activities because of time constraints.
5. Adaptive	Scientific software tends to evolve due to new experimental results, model development and continual augmentation of functionality [16]. Thus, the requirements and changes are extremely dynamic in scientific-software development. Likewise, succeeding testing on scientific software is problematic due to lack of elicitation and specification of requirements in the early phases [10][19].
6. Incremental	New features that come from new findings might be added to the system. The features of the system are built in small steps.
7. Convergent	Although, the developers do not know the details of all requirements in advance, they try to reduce the risk by writing code based on necessary features. They add new features into the system iteratively, until the system meets the customers satisfaction.
8. People-Oriented	Strict planning may not be suited to scientific-software development because scientists needs the flexibility to perform the experiment as quickly as possible [5].
9. Collaborative	Developers might not thoroughly understand the problem in the scientific domain. Therefore, developers and scientists need to work constantly together with close communication.

II. Case Study

A. Research Procedure

This case study investigates the level of support that Agile methods provide to scientific-software development. The focus question we aim to address is *"How does the Agile method support the scientific software-development process?"* The case study investigates the scientific-software development of the CLiiME project (Community Laser-induced Incandescence Modeling Environment). This project was developed in the Combustion Research Facility at Sandia National Laboratories.

In its current stage the CLiiME project has three developers (Nanthaamornphong, Morris, and Rouson) and one domain expert (Michelsen). All developers have more than five years of experience in software development, but they have never been involved in an Agile software-development process. As part of this research process we have collected materials and held frequent discussions with the stakeholders throughout the life of the software-development project. The collected materials consist of the project plan, software designs, collaborated emails, and other materials created within the project. The theme of the discussions includes not only software requirements and specifications but the impact of the Agile practices in this project. The first author documented the discussions.

One of the interesting facts about this case study is that CLiiME was developed as an open-source software, and it will be distributed to scientists, including scientific-software developers and non-developers. The LII model used in this software project is still under active development. Under this dynamic environment an Agile development approach is absolutely necessary. Furthermore, CLiiME was the first project that was developed under the Agile method at the Combustion Research Facility.

B. Description of CLiiME

The Community Laser-induced Incandescence Modeling Environment (CLiiME) aims to predict the temporal response of laser-induced incandescence (LII) from carbonaceous particles based on the exact model [12]. The model accounts for particle heating by absorption of light from a pulsed laser and cooling by sublimation, conduction, and radiation. The model also includes mechanisms for oxidation and annealing of the particles and non-thermal photodesorption of carbon clusters from the particle surfaces. CLiiME allows users to customize the model with their own equations. In addition, CLiiME provides the tools to visually compare the results of the simulation with experimental results. The overall structure of CLiiME is illustrated in Figure 2.

CLiiME was implemented with two object-oriented programming languages, Fortran and Java. The model part (LII-Model), which is responsible for all calculations within the model, is implemented with object-oriented Fortran. Whereas, LII-UI, the graphic user interface (GUI) of the system, is implemented with Java. We designed the model to support an LII simulation for a particle affected by the energy mechanisms

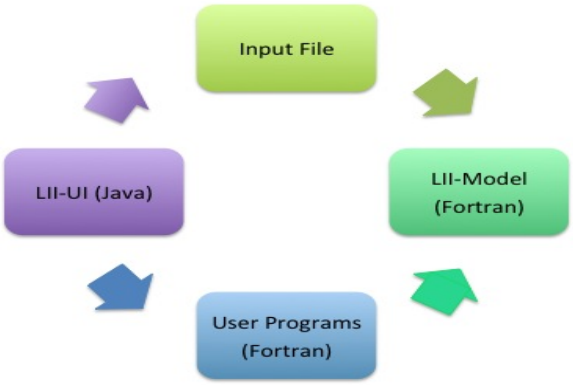

Fig. 2. Overall Structure for CLiiME

explained by Michelsen [12], but other energies might be added in the future. The LII-UI allows users to customize the energy mechanisms for their own models. Based on the current version of the software, the model supports the following mechanisms for a particular simulation:

1) Absorption
2) Annealing
3) Conduction
4) Radiation
5) Oxidation
6) Sublimation
7) Scattering
8) Extinction
9) Sublimation extension
10) Thermionic

In its most recent release the model included the following general capabilities:

- Track the evolution of a single particle. The evolution of a collection of particles is assumed to be a multiple of the contributions from a single particle. Aggregates are accounted for by surface area modifications to the contribution of a single particle.
- Allow the user to modify input parameters, such as the absorption coefficient, accommodation coefficient, density, etc.
- Allow for the extension of the model to deal with a collection of particles of different sizes.
- Enable capabilities for curve fitting simulation data.
- Generate output data in a simple text file format.
- Accept user-provided flags to set the energy mechanisms that are part of the model used for a particular simulation.
- Accept user-provided laser wavelength, fluence value, detection wavelength and detector wavelength response function, and any other parameter required by the specific model.
- Accept a user-provided input file with the laser temporal profile for the simulation.

In the GUI, the current version of CLiiME supports the following features:

- Allow the users to modify the energy mechanisms or write their own energy mechanisms.
- Allow the users to build the script for compiling and running their own modified model.

- Allow use of graphing interface to view the generated output.

C. Agile Practices

This section describes the Agile practices used throughout the project. We attempted to follow the rules of eXtreme Programming (XP) during the project. The philosophy of XP is to make the customer satisfied with the software product. Taking the customer onto the team and receiving feedback frequently are the approaches to achieve the goals of XP. The feedback of customers should be taken into account throughout the software development.

The lifecycle of an XP project is shown in Figure 3. The XP process is divided into six phases: Exploration, Planning, Iterations to release, Production, Maintenance, and Death [3]. In the *Exploration* phase, the customer writes the stories included in the system. The set of written stories are prioritized, and a schedule of the first release is developed during the *Planning* phase. The architecture of the whole system gets created in the next phase, *Iterations to Release*, which includes continuous testing and code integration. Before releasing the system, developers may need to perform additional testing in the *Production* phase. In the *Maintenance* phase, any suggestions about the system are documented for later implementation in the updated release. Finally, the *Death* phase occurs when there are no more stories or changes from the customer.

In general, development under XP practices is done iteratively, and phases sometimes overlap; thus it is not easy to provide a broad step-by step description of the XP practices. In the following discussion we further describe XP practices including both practices we implemented and did not implemented. Table II lists 29 rules of XP[1]. The last column in the table indicates whether the practice was followed during the development of CLiiME or not.

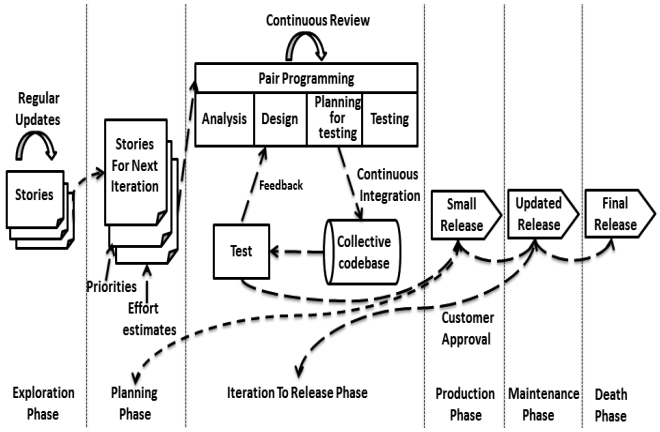

Fig. 3. Lifecycle of the XP process (adapt from [1])

1) User stories are written. Since the model was still under active development, we gathered the requirements through a published journal paper rather than written documentation of requirements. We used the combination of short development iterations and stakeholder feedback to decide the details of specific features.

2) Release planning creates the release schedule. A project plan was created in the beginning of the project. The project plan included high-level goals and a schedule for the project. We have planned the first release of the software. The schedule was specified along with the description of each task.

3) Make frequent small releases. Based on the schedule, we had the target of having a small release every week. The developers and stakeholders agreed on a date for each release of the project. The stakeholders chose the stories that they wanted to be done first. Table III shows all releases in this project.

4) The project is divided into iterations. We divided the development schedule into a number of iterations. Each iteration took one to two weeks in length. Every two weeks, the stakeholders were asked to evaluate the progress and performance of the project. They also provided their feedback in terms of new user stories. As a result, stakeholders could predict how quickly the team was progressing and could

[1]http://www.extremeprogramming.org/rules.html

TABLE II
AGILE PRACTICES

No.	Practices	
1.	User stories are written.	x
2.	Release planning creates the release schedule.	x
3.	Make frequent small releases.	x
4.	The project is divided into iterations.	x
5.	Iteration planning starts each iteration.	x
6.	Give the team a dedicated open work space.	-
7.	Set a sustainable pace.	-
8.	A stand up meeting starts each day.	-
9.	The Project Velocity is measured.	x
10.	Move people around.	-
11.	Fix XP when it breaks.	-
12.	Simplicity.	x
13.	Choose a system metaphor.	-
14.	Use CRC cards for design sessions.	-
15.	Create spike solutions to reduce risk.	x
16.	No functionality is added early.	x
17.	Refactor whenever and wherever possible.	x
18.	The customer is always available.	x
19.	Code must be written to agreed standards.	x
20.	Code the unit test first.	x
21.	All production code is pair programmed.	-
22.	Only one pair integrates code at a time.	-
23.	Integrate often.	x
24.	Set up a dedicated integration computer.	x
25.	Use collective ownership.	x
26.	Acceptance tests are run often and the score is published.	x
27.	All code must pass all unit tests before it can be released.	x
28.	When a bug is found tests are created.	x
29.	All code must have unit tests.	x

TABLE III
PLANNED RELEASES

Release	Delivery Product
1	Core framework include: Design and all interface components
2	Main energy objects include: Annealing and Conduction
3	Main energy objects include: Absorption and Oxidation
4	Utility functions
5	Mathematic functions
6	Other relevant energy objects include: Coating, Extinction, Scattering, Sublimation Extension, and Thermionic
7	Graphical User Interface : Customized energy builder
8	Graphical User Interface : Script builder and Gaussian builder
9	Graphical User Interface : Report viewer

schedule high-priority stories early. We found that frequently-delivered software in each iteration enabled gathering feedback that was used to guide the next iteration. Also, the meeting in each iteration kept developers up-to-date on the project's status.

5) Iteration planning starts each iteration. Before starting a new iteration, we had a meeting. We divided the *tasks* and distributed the tasks to developers. We used the velocity to determine how the tasks should be divided.

6) Give the team a dedicated open work space. We had limited work space and few developers involved in this project, so we did not set up the work space according to the XP guideline. The XP guideline suggests that computers should be in a central area that no one owns. Similarly, the XP suggests that a conference table provides the place for group discussions. In this project, we used an office room as the conference room.

7) Set a sustainable pace. Although we did not set the number of working hours, each developer regularly worked on the project for 8-9 hours per day. In some cases, developers have worked for over 9 hours if the progress was behind schedule.

8) A stand-up meeting starts each day. Instead of setting up daily meetings, developers were encouraged to meet with others when they needed, so the communication between team members was rather straightforward. Moreover, we had weekly meetings in which relevant topics, such as the progress and plans, were discussed. In addition to the weekly meetings, we also set up meetings when a developer encountered a technical problem. To schedule a meeting, the developer who actually needed the meeting had to make the invitation to all attendees via e-mail. The meeting was setup when all attendees responded to the invitation. Besides face-to-face meetings, e-mail was often used to communicate with stakeholders.

9) The project velocity is measured. Every week we measured the number of stories, known as *velocity*, which we completed. We used the *velocity* to predict how much work would be completed in later weeks. In addition, we used the *velocity* to plan the next iteration. The stakeholders were frequently informed about the progress of the projects so they were able to predict how quickly the team would add new functionality.

10) Move people around. As we have only a few developers in this project, we did not move developers around the code.

11) Fix XP when it breaks. During the software development of this project, we found that the process was well-followed. All developers understood and strictly followed the process.

12) Simplicity. Developers always tried to find the simplest possible design for stories. With each iteration, developers improved the design of the system. Developers identified the complex part in the software design and replaced them by a simple one. For example, we maintained the configuration data with flat files rather than using a database. We avoided making a system without multithreading. In this project, developers implemented the model with Fortran 2003, which supports full OO features. Based on the OO paradigm, an OO programming language focuses on creating objects rather than creating procedures like a procedure programming language (e.g., Cobol, C, Fortran 77). Object-oriented features promote data and code separation, which prove useful when addressing complexity within an application [6] . Moreover, OO design allows developers to reuse pieces of code. All these OO characteristics might support practitioners to accomplish maintenance tasks [15].

13) Choose a system metaphor. The metaphor is used to describe an overall picture and common understanding of the system and its functionalities. We did not explicitly have a system metaphor in this project. However, to make sure that all developers understood how the system works, all developers had to read the relevant paper about the LII models. In addition, a model developer (the author of the journal article [12]) explained the mechanisms of the model and how the model works. Software developers also had the chance to interact with the model developer to make sure that they understood the system in the same direction.

14) Use CRC cards for design sessions. Rather than using class responsibility collaboration (CRC) cards for designing the system, we used Unified Modeling Language (UML) diagrams as a tool to represent the design of the system. UML is a graphical notation for drawing diagrams of software concepts. Developers can use it for drawing diagrams of a problem domain or a proposed software design. The UML can be used to describe concepts and abstractions that exist in the problem domain. In addition, UML is convenient for communicating design concepts among developers. In fact, UML consists of many types of diagrams (e.g., Use case, Class, Sequence, State). In this project we mainly used class diagrams to represent the major classes and relationships in the system.

Extracting UML diagrams from existing Fortran source code could prove a useful tool to determine if the written code conforms to the design. Nanthaamornphong has developed a tool to generate UML design documents from OO Fortran code. The tool is called *ForUML*[2]. The current version of

[2]http://aziz.students.cs.ua.edu/foruml.htm

ForUML can automatically parse Fortran source code and create the UML class diagrams; a detailed discussion of ForUML will be the subject of a future publication. Based on the UML diagrams developers within the project modified the code that did not agree with the design. During development, developers compared the class diagram obtained from ForUML with the original UML class diagram created manually. After comparison, they determined whether the code was breaking the design. Instead of inspecting the source code directly, developers were able to make the comparison/decision with less effort. The UML class diagrams, were also used by developers as part of the progress report.

15) Create spike solutions to reduce risk. We often created spike programs when we found a problem during development. One example technique that we used was to find the tutorial of a similar problem on the internet. In other cases we developed small program test cases. For example, we encountered a problem reading name lists in Fortran; name lists are groupings of key and value pairs. We observed different reading mechanisms in different implementations of the Fortran compiler. We conducted experiments with a small program using different Fortran compilers; we found that both the order of the name lists and the order of the list of values within each name list affected the input values.

16) No functionality is added early. We did not implement unnecessary functions or components in the system. We often reviewed the code and eliminated unnecessary functions. We only focused on what was scheduled for each release.

17) Refactor whenever and wherever possible. We used refactoring frequently to evolve the design and improve the maintainability of the software. Refactoring and testing facilitate development of code that is easy to modify. We refactored the code to keep it simple and eliminated unused functionality.

18) The customer is always available. In this project, the customer did help developers move forward. The customer quickly responded when the team needed to get answers about the system, so the developers gained a deeper understanding of the system. The customer was involved in the project before starting the project and at every iteration. The customer participated in scheduled team meetings to make the plan and provide useful feedback.

19) Code must be written to agreed standards. The coding standard is the set of rules and recommendations for coding in a particular programming language. XP does not provide further instructions on how to create a coding standard. Therefore, we defined a coding standard for the Fortran implementation before the project was started. Below are the objectives of our coding standard and examples.

- Reduce compilation errors and increase software portability. For example, the Fortran coding standard defines a maximum line length (132 characters); obeying standard rules allows developers to circumvent portability problems. In the coding standard we recommend developers use spaces instead of tabs because different environments and editors interpret tab characters in different ways.
- Help developers understand the code. For instance, the standard recommends developers identify variables, procedure names, function names, and file names descriptively. To the extent that it is possible, unnamed constants must be avoided; all values must have a variable name.
- Conform with other object-oriented programming languages. One of the coding standard rules for Fortran recommends developers use the name "this" for the *pass* argument in all type-bound procedures and the dummy argument of the type in all module procedures. In many object-oriented programming languages (e.g., Java, C#, C++), "this" is used within a function to refer to the object by which the function was called. In Fortran, the object passed to the type-bound procedure has the *pass* attribute. This rule will help future developers who are familiar with other object-oriented programming languages easily understand the Fortran source code.
- Reduce the communication problem on a team. The coding standard guides developers to add comments to their code to allow other developers to understand its purpose. In this project the comments include the necessary constructs for the generation of doxygen[3] documentation. Otherwise, developers should avoid providing unnecessary comments, which will make other developers confused.

For coding standard in Java, developers followed the "Java Code Conventions"[4] document, which is currently maintained by Oracle.

20) Code the unit test first. We deployed the Test-Driven Development approach to develop the system. We will discuss this practice in the next section.

21) All production code is pair-programmed. In this project, we used pair programming when reviewing the code. Before releasing the code, all code has been reviewed by two programmers at a single computer. Two programmers sat side by side in front of the monitor. Both programmers concentrated on the code being reviewed. We found that pair programming helps speed up solving relevant issues. When two developers sat in front of a computer, they were more likely to concentrate on a specific problematic feature rather than doing irrelevant tasks.

22) Only one pair integrates code at a time. Since we used the source code repository to control integration in the project, every developer could integrate code at anytime. However, we found a small number of conflicting file problems, discussion among developers resolved those problems.

23) Integrate often. XP suggests that XP teams integrate their accomplishments more often because the continuous integration constantly reduces the chance of conflicting changes. In this project, developers often submitted the code into the code repository several times a day. Every developer needed to work with the latest version.

[3] www.doxygen.org/

[4] http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

24) Set up a dedicated integration computer. We have one server machine to control the source code and build the system. Developers needed to pull the latest version from this server. To control the release versions in this project, we used Git[5] as the version control system. Git allows a team of developers to work on the same set of files without interrupting other developers. In addition, Git can track the evolution of code, so we can see the log of each changed file to analyze how it evolved and who did it.

25) Use collective ownership. Any developers under this project could modify or improve the code based on unit tests. No developer was responsible for any one particular functionality. Similarly, no developer had more authority than another over a system. However, developers must follow the rule that the code must pass all tests before submitting the code into the repository. In the opinion of the authors, this practice can be especially useful in the case of large complex scientific software because it is impossible that every developer would know the code well enough.

26) Acceptance tests are run often, and the score is published. Acceptance tests were written by the customer. The customer verified the correctness of the acceptance tests, which represent expected results from the system. After running the acceptance tests, the team was informed of the test score in order to fix any failed tests. In this project, we considered the acceptance tests as the user requirements.

As the rest of our practices are related to the testing and TDD, we will discuss them in the next section.

D. Test-Driven Development

Testing plays an important role in XP. As we mentioned earlier, this project employed a TDD approach. Next, we describe the practices in TDD we performed in this project.

1) Write a test: When developers write a test in TDD, they are actually making design decisions. In this project, developers focused on how each functionality can be implemented and then developed a unit test. Unit tests are tests that verify that small elements of the system are working as expected. The unit test probably would not compile at this point. Therefore, developers need to create a concrete implementation for the functionality.

The challenge in this step is to write a good unit test. A unit test should test only very specific functionality. If the unit tests are rough, a test failure is more difficult to discover. On the other hand, if the unit test exercises a small amount of code, the test failure can be spotted very quickly. To test the partial differential equations (PDE) modeled in this application, we tested the output of different functions involved in the PDE (e.g., Runge-Kutta method). We compared the obtained results with expected results computed with another commercial software application, called *Igor*; which provided some built-in functions involved in the PDE calculation. More specifically, we compared those results for a small number of time steps (e.g., 10 steps). In this test case, we compared the results of each time step obtained from CLiiME with expected results produced by Igor. The debugging capability of Igor was used to acquire the expected result of each time step. Subsequently, other test cases were created to check results of the whole time evolution. Throughout this project, we have set an error tolerance value (the difference must be less than 10%) for avoiding the round-off error problem when verifying the computed values. To establish consistent testing, all developers had to use a developed function employing the tolerance value to compare two numerical values. Although our testing strategies allow developers to validate the output with confidence, we might say that to develop unit tests in scientific software projects requires additional techniques to identify the appropriate testing task required for a given development environment. Another problem when writing a unit test is the unit test dependency. Problems arise when one unit test depends on another unit test. To solve such problems, developers tried to minimize the dependency between unit tests. In addition, developers followed these general rules when creating the unit tests:

- Tests should be isolated and order independent (atomic).
- Tests should be run fast.
- Test should not require manual setup.

In addition to write a test for the particular functionality, developers also created a test when they found the bug.

2) Write the code to pass the test: The next step is to write just enough code to make the test pass. The main goal of this step is to make the test pass as quickly as possible. Developers wrote the code to meet only the current requirements and did not try to predict what will be included in the future.

3) Refactoring: Martin Fowler defines refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [7]. The main purpose of this step is to make the code better. Refactoring helped developers to maintain the software easily.

The main characteristic of scientific software is that requirements are often changed. Those changes in the existing code should be minimized. The *Open/Closed principle* states that redesign and new written code should be added with minimum changes to the existing code. The design should allow the addition of new functionality and new classes without modification of existing code. Conformance to the Open/Closed principle is what yields the benefits claimed for supporting the reusability and maintainability of the system. In order to achieve this principle, we used object-oriented Fortran and required the developers to apply abstractions to parts of the system that exhibit frequent change. Key mechanisms that support the Open/Closed principle are abstraction and polymorphism. Thus, developers tried to incorporate design patterns into the software design.

A design pattern is a particular software design solution for solving a problem that can be used repeatedly in similar conditions in the future. Design patterns can help both novice and experienced software designers identify and solve software design problems. Two design patterns we used are *Strategy*

[5]http://git-scm.com

and *Factory Method* patterns. The *Strategy* pattern solves the problem of inverting the dependencies of the generic algorithm and the detailed implementation in a very different way. This pattern allows users to select the approach to calculate differential equations (e.g., Runge-Kutta, Euler). The *Factory Method pattern* creates objects to implement an algorithm. A superclass specifies all standard and generic behavior, and then delegates the creation details to subclasses. This pattern allows developers to implement different types of energy objects.

Currently, many tools support unit testing. One of the most common types of unit testing tools is a unit test framework. In this project, we chose two frameworks to perform unit testing:

- CTest Framework[6]
- JUnit Framework[7]

CTest is part of the CMake open-source system that manages the build process of a software application using a compiler-independent method. CTest helps developers define the unit-test code, control the execution of tests, run the tests, and report the results of tests through a single command. We used CMake/CTest with the Fortran code and ran the same build process on various Fortran compilers (e.g., GNU, Nag). For Java we used JUnit, an open-source software that became the de facto standard framework for developing unit tests in Java. JUnit has many features that allow developers to write and run tests.

The benefit of using these frameworks is that test cases are easy to automate. When the code was changed, developers ran all tests to identify the effects of the change. If any test failed, developers needed to correct the code until all tests passed again. Both unit testing frameworks have a similar process for TDD. The first step is to write the unit test code for a new feature (or changing an existing one). In general, the tests will show that the test case is not passed by not compiling or failing when executed. Once a developer implemented a feature, the test passed. Before modifying any code, developers first ran all unit tests. Our rule in this project was that all unit tests in the repository must pass, so developers had to check to make sure all tests passed before coding. However, a main difference between the two frameworks under this project is that JUnit was integrated in the Netbean IDE[8] used to implement the GUI, whereas CTest was used by developers manually without any IDE.

III. Discussion

We draw several conclusions from this study. First, Agile practices provide methods to address risks early in the development cycle. Second, the Agile method appears to be particular well-suited to scientific software projects. Third, TDD helps developers to avoid problems concerning the software quality by driving the development with sufficient tests. Further details are discussed below.

1. Agile practices provide the methods to address risks early in the development cycle. The most significant risk factors in the software-development process are responding to change. The Agile method can support handling the changes, such as continuous feedback from the customer and short iterative development. Agile practices allow developers to develop the software without gathering all requirements in the early stage. Developers do not need to identify all the required features of the system before starting the project, unlike traditional software-development life cycles where well defined requirements are established before implementation. Moreover, feedbacks from stakeholders, particularly the customer, during the software-development process contributes to the software meeting customer satisfaction.

2. Agile practices appear to be particular well-suited to scientific software projects. Scientists often discover new solutions to address underlying problems; as a result the model of the problems continually changes. For example, the scientist found that the number of particles might affect the energy with different mechanisms result in additional energy mechanism might be involved in the model. Agile practices might properly fit this situation. New features in scientific software are acquired iteratively as context. To use the Agile development process, developers are always ready to embrace such new features into the system. Also, the frequent demonstration and release of software in Agile practices enable customers to see details on the current release and thus provide feedback to refine the requirement. As a result, developers eventually present customers with a system containing all desired features. This conclusion conforms to the results of Sletholt et al's work [19], which they investigated several scientific projects implementing Agile practices. They also mentioned that the Agile is suitable for scientific projects, which has a small team.

3. Test-Driven Development helps developers avoid problems concerning software quality. TDD guarantees the software quality from the beginning. Developers are encouraged to write only the functionalities that make all tests pass and thus meet the requirements. In addition, when developing the software with testability in mind, the result is that the software is extensible, flexible, and maintainable. Also, developers do not worry about an aspect of a feature that is not in the tests. The primary goal when writing code with TDD is simply to make the tests pass. In order for software to evolve by demands, the software should support new augmented functionalities with little cost. The ability to support such changes is very important, especially the maintainability. Refactoring for maintainability should work to simplify the code wherever possible. The refactoring process might be comprised of making the methods short, simplifying control structures, and providing descriptive identifier names (e.g., function, variable, and class names). Other benefits of refactoring are:

- Easy to add new code - Developers could add new features and refactor later without regard to how well a new feature fits with the existing design.
- Improve the existing code - We continuously improved the software design with refactoring. Continuous refactoring involves constantly detecting poor design. In addition,

[6]http://www.cmake.org
[7]http://ww.junit.org
[8]http://www.netbean.org

developers refactored the code to eliminate duplication and to clarify the code.

4. Lessons Learned. The lessons learned from this case study might be useful to software engineers and scientists:

- **Agile Awareness** One of the most difficult tasks involved with using Agile practices is successfully introducing them to the team. All developers in this project had little prior knowledge about Agile. In the early development phase, developers were not fully aware of the benefit of the Agile method. The lack of awareness leads to doubtfulness about Agile practices. As a results, developers were not interested in the practices. For example, TDD requires developers write about the same amount of test code as actual code. Obviously, developers acknowledged the extra work needed to be done, but did not see the benefits of TDD in the beginning of the project. Fortunately, developers were exposed to the benefits of the TDD while working on the project. For example, one developer who has a little knowledge about the scientific domain reported that he could write the actual code easily because he knew the expected results from the testing code. Another example, a developer informed that the refactoring code helped him reduced the number of Type and number of Type-bound procedure in many modules. Subsequently, he could quickly made the documentation for those modules. However, we believe that it would be useful for developers if there were training sessions about the Agile before starting the project. The main goal of these sessions should be to provide the knowledge about the Agile method and its benefits. In addition, having a mentor who knows Agile development well can help developers come up to speed significantly faster.
- **The tool for TDD** The TDD method relies on the set of tests that are continually executed during development. It would be helpful to run the test suite automatically without too much effort. Currently, there are some useful tools for performing the TDD and those tools are designed for different purposes. Therefore, the team should carefully consider which tools would be most useful for the working project. Although this project did not use other tools for handling the requirements or bug reports, the authors are of the opinion that, as the software becomes more complex and the project involves other parties, the software tools would be necessary.
- **Tailored Agile practices** Although Agile practices seem to fit scientific-software development better than traditional software-development practices, there is still a need to adjust Agile practices according to the nature of the scientific-software project. For example, our project did not use the pair programming regularly. We used the pair programming in an ad hoc manner, e.g., when developers faced a technical programming concern, or when reviewing the code. Some other similar projects that have a small group of developers might use paired programing when confronting different circumstances.
- **Refactoring strategies** One of the techniques we used in our refactoring process was the use of *Design Patterns* [8], and the subsequent assessment of improvement in the design because of design patterns. In this project, developers did not only use ForUML to compare the code and its design; the tool was also used to identify existing design patterns or candidates for new design patterns from written code. After the design was identified, the impact of design patterns on the system was evaluated. If the system was not improved, we refactored with another approach. Furthermore, developers must be carefully of problems that arise when using design patterns: increased code complexity and unnecessary design patterns. Unfortunately, experience is the key component we have identified to prevent those problems. A good understanding of the purpose of each design pattern will minimize the problems.
- **Documentation of the system** The project documentation in the Agile method is an issue that has drawn a lot of attention. The common question about the documentation is "how much of documentation is enough?" In general, customers demand more documentation than needed, and that documentation is not well-produced [18]. Although the Agile method does not stress the need for documentation, we found that documentation was needed in order to preserve critical information over time and support the maintenance tasks. Therefore, developers need to document code that might raise a problem in the future.
- **The challenge in Agile Practices** Since we did not measure the code defect and maintenance cost in this project, we cannot conclude whether the Agile practices increase more maintenance cost than previous scientific projects developed at the Combustion Research Facility. In fact, the current version of the CLiiME contains 3,378 LOC for actual code and 2,160 LOC for testing code. On the other hands, we might say that 40% of all written code is the additional code, which needs to be maintained at a later stage as well as the delivered program. In addition, we report results of this project during the 4-month life span of the project. For generalizability, the most scientific projects might have a life span of several years, implying that the amount of testing code will also increase. Therefore, to reduce the amount of testing code and maintenance cost provide the opportunity for fruitful research. Although the overview of the Agile has positive effects on this project, we did not evaluate the quality of the software development process. The quality evaluation can help the team improve the process and identify the problems at the early time. For example, at the beginning of this project, the structure of directories in the integration server was improperly arranged. As the result, we were facing with the dependency problems. After reorganizing the structure, the problem was resolved. We believe that the quality assessment can identify those problems in advance.

IV. Conclusion

This paper discusses a case study of the scientific software-development project, called CLiiME sponsored by the Sandia National Laboratories, a US Department of Energy laboratory. The goal of this paper is to provide information that can be useful for the software-engineering community as well as the computational science and engineering community. For the software-engineering community, this paper highlighted some of the reasons why the scientific software-development process is different from traditional software-development process. For the computational science community, this paper provided some practices that can improve the scientific software-development process.

The adaptation of Agile development methods has increased significantly in various software domains. This case study demonstrates some of the benefits of the Agile method to the scientific-software development. Agile practices are well suited to the exploratory, iterative, and collaborative nature of scientific inquiry, especially when considering theory development through scientific experiments. We have identified Agile practices adopted in CLiiME, and we also identified the benefits of using these practices as followings:

- The Agile process mitigates the risks including unclear requirements, new features during software development, and technical problems.
- The system was frequently evaluated and improved through continuous integration and small iterations. As the result, the software quality is increased.
- The Agile method reduces the developers' effort. The waste of unnecessary work (e.g., requirement documents, unnecessary features) is reduced. The developers only focus on actual work.
- The customer and stakeholders can monitor the progress and provide the feedback during development. The feedback helps developers get better insights in the system. In addition, to integrate customer reviews more often increases satisfaction with delivery.

When compared to the traditional software-development approach, the strong conformity between the Agile method and the scientific method indicates that the traditional approach is insufficient. The project is typical of many scientific-software projects in the field today, and we believe that these experiences can be applied to other similar projects.

As we conducted only one case study, however, it limits the extent to which the findings can be generalized. Various case studies would be more likely to provide results that could be generalized. In the future, we have a plan to conduct case studies with different scientific-software projects, especially with projects that have more developers, involving other parties (e.g., subcontractors, other organizations).

Acknowledgments

This work was funded by the Division of Chemical Sciences, Geosciences, and Biosciences, the Office of Basic Energy Sciences, the US Department of Energy. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the National Nuclear Security Administration under contract DE-AC04-94-AL85000.

References

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods - Review and analysis," VTT PUBLICATIONS, Tech. Rep. 478, 2002.

[2] K. Beck, "Emergent control in extreme programming," *Cutter IT Journal*, vol. 13, no. 11, pp. 22–25, 2000.

[3] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[5] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 550–559.

[6] V. Decyk, C. Norton, and H. Gardner, "Why fortran?" *Computing in Science Engineering*, vol. 9, no. 4, pp. 68 –71, july-aug. 2007.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] P. Gilbert, *Software Design and Development*. USA: SRA School Group, 1986.

[10] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, ser. SECSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/SECSE.2009.5069155

[11] L. Koskela, *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Greenwich, CT, USA: Manning Publications Co., 2007.

[12] H. A. Michelsen, "Understanding and predicting the temporal response of laser-induced incandescence from carbonaceous particles," *The Journal of Chemical Physics*, vol. 118, no. 15, pp. 7012–7045, 2003. [Online]. Available: http://link.aip.org/link/?JCP/118/7012/1

[13] H. A. Michelsen, F. Lui, B. F. Kock, H. Bladh, A. Boiarciuc, M. Charwath, T. Dreier, R. Hadef, M. Hofmann, J. Reimann, S. Will, P. E. Bengtsson, H. Bockhorn, F. Foucher, K. P. Geigle, C. Mounaïm-Rousselle, C. Schulz, R. Stirn, B. Tribalet, and R.Suntz, "Modeling laser-induced incandescence of soot: a summary and comparison of LII models," *Applied Physics B*, vol. 87, no. 3, pp. 503–521, 2007.

[14] G. G. Miller, "The characteristics of agile software processes," in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, ser. TOOLS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 385–387.

[15] P. Ramachandran and M. Ramakrishna, "An object-oriented design for two-dimensional vortex particle methods," *ACM Trans. Math. Softw.*, vol. 36, no. 4, pp. 18:1–18:28, Aug. 2009.

[16] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *Software, IEEE*, vol. 25, no. 4, pp. 21–28, July-Aug 2008.

[17] C. Schulz, B. F. Kock, M. Hofmann, H. Michelsen, S. Will, B. Bougie, R. Suntz, and G. Smallwood, "Laser-induced incandescence: recent trends and current questions," *Applied Physics B*, vol. 83, no. 3, pp. 333–354, 2006.

[18] A. Scott. (2002, Jul.) Duking it out. [Online]. Available: http://www.drdobbs.com/architecture-and-design/duking-it-out/184414875

[19] M. Sletholt, J. Hannay, D. Pfahl, and H. Langtangen, "What do we know about scientific software development's agile practices?" *Computing in Science Engineering*, vol. 14, no. 2, pp. 24 –37, March-April 2012.