Extreme Programming: A University Team Design Experience

Jeremy Kivi, Darlene Haydon, Jason Hayes
Ryan Schneider, Giancarlo Succi[§]

Department of Electrical and Computer Engineering
University of Calgary, Canada

[§] Department of Electrical and Computer Engineering
University of Alberta, Canada
Contact E-mail: Giancarlo.Succi@enel.ucalgary.ca

Abstract

This paper discusses an experience in applying the extreme programming approach to the 4th year team design project course. Extreme programming is a recent methodology for software system development that focuses on high customer integration, extensive testing, code-centered development and documentation, refactoring and paired programming. Typically, the project course is managed using the standard waterfall or V-shaped development models with a faculty advisor acting as a customer for the project. In this project extreme programming has been used instead.

Extreme programming is based on a sequence of development practices, including pair programming, very accurate configuration management, strong customer interaction based on "system stories," detailed testing. In this project, paired programmers are used for the duration of a release and then the pairs rotate. The distributed programming environment is handled using the JCVS suite of configuration management tools. Every 3-4 weeks, a new fully functional release is delivered and reviewed by the customer. The specifications for each release are captured incrementally using use case scenarios. Only the essential requirements for the current iteration are implemented. The JUnit test suite is also used to test each of the Java classes on an ongoing basis. The test suite verifies all aspects of the software at each build; this is necessary when refactoring components.

Requirements capture, design and implementation of the deliverables are performed incrementally and result in quicker development times and reduced defects. Refactoring is applied wherever possible to simplify the code. Documentation is applied using the standard JavaDoc utility and is kept to a minimum. Finally, customer feedback is immediately incorporated into future iterations of the design process.

Most of the features of extreme programming have worked well. Difficulties are found in applying some principles in the context of a university course. Schedules, team member programming habits, lack of experience and preferences are conflicting with the paradigm.

1 Introduction

A 4th year team design project group at the University of Calgary has adopted the Extreme Programming (XP) paradigm in an attempt to develop and deliver the best possible software product to their customer. The team consists of four students, three members with no prior Java experience and one member with some prior experience. Java 1.2 is the primary development language at the customer's request. The team has no prior experience with the XP process.

The following paper is a discussion of the implementation of XP for the team design project, in a university environment. The discussion is presented with one caveat. The members lacked experience in both XP and software development, at the start of the project. In fact, several of the above XP concepts were implemented several months after starting. This has severely changed the dynamic of development and the XP paradigm. If this team were now given a new project, the outcome of the project and implementation of XP would be more positive.

1.1 Domain of the Project

All 4th year students are required to complete a 4-member team design project, in the eight months of their final year. This team's project is to design and implement a statistical analysis tool for software metrics. Another tool, WebMetrics is a client-server application with various parsers for collecting software metrics. The team's statistical analysis tool is intended to compliment the tool suite already present.

The statistical analysis tool is required to provide first-order, descriptive statistics, bootstrap estimates of standard errors and confidence intervals, perform meta-

0-7803-5957-7/00/$10.00 © 2000 IEEE

analysis and other parametric and non-parametric tests. The primary use of the tools is to depict relationships in the software metrics data provided. Also, additional correlations and statistics can be done between metrics collected by the Webmetrics tool suite, and financial or managerial data collected in parallel with the software development process.

2 Extreme Programming Aspects

The following items relating to XP will be discussed: paired programming, refactoring, incremental deliverables, use case requirements capture, unit testing and configuration management.

2.1 Paired Programming

Paired programming is touted as a very useful development practice. This concept encourages developers to code review (each other) all the time and removes some of a programmer's bias towards his or her own code. In the case of high algorithmic or information complexity, paired programming helps developers to better understand the project as they design and develop code together. Highly experienced and valuable team members can be paired with new, or inexperienced members. This allows the transfer of knowledge and experience throughout the project team, between pairs. Finally, this socialization process minimizes the need for extensive documentation.

Having made the above statements, four months into the project the team discontinued this practice. Team members felt that paired programming was a waste of time. They cited the following reasons: Scheduling between pairs in a university setting was extremely difficult. Unlike industry with a set of common hours, partners' schedules were never aligned for extensive periods of time. In the event that pairs were programming known or accepted code, like standard data structures and algorithms, there was no sense in having two people. Finally, there were no experienced developers to pair up with and learn from.

2.2 Incremental Requirements Capture and Deliverables

Another aspect of extreme programming that our team has implemented is incremental deliverables. At the start of the project our team sat down with our customer and we broke the project into 4 different pieces. Each piece in themselves would be a mini project with it's own use cases and mini milestones. We would start with one section and determine a deliverable date, set up use cases and then develop a plan to meet that date. Time between releases is between three to four weeks. Each deliverable would be a fully functional piece of the complete project. With each increment we would add functionality to the already existing prototype.

We have found that this process has helped us focus our efforts, has allowed our customer to provide added feedback and to see progress. Our focus and the project have become very customer-centric; this helps to ensure the success of the project.

By focusing on one area we have been able to quickly build a working prototype without being overwhelmed by the entire scope of the project. With each new set of requirements we are able to nail down what exactly is expected. We are also able to see our customer's reaction to the working versions throughout the course of the project rather then just at the end. This has allowed us to make the necessary changes to ensure our customer is happy.

Our team however is rather new to project management and is also new to the language we are using to develop. This has presented several problems within the first increment. One of the requirements was not fully understood by the team and this has caused us to spend more time refurbishing our code. However we are grateful that due to a mini deliverable we were able to catch this problem relatively early in the project as opposed to at the end. The changes that needed to be made were less significant now then they would have been at the end of the entire cycle.

Another aspect of the incremental process that our team is having difficulty with is the lack of planning for the future. You start with the first mini deliverable and you architect your code based only on the requirements for that one section. Now that we are onto the second increment we are finding that we are spending a lot of time changing the architecture so that we can implement the next phase. It is our belief that if we had spent a little more time at the start looking at all the pieces of the project we would not have had this problem.

Overall we believe that this method is very useful for producing a functional piece of software that meets the customers' requirements within a short amount of time. We feel that although the first increment presented us with several problems that we will be able to use our experience to ensure that the remaining increments run smoother.

2.3 Use Cases

Use cases are a very useful way to capture customer requirements to ensure that both the customer and the team understand the functionality. At the start of the project our team sat down with the customer to determine the general requirements of the entire system. From the information gathered in that meeting and using UML, we were able to write out a set of use cases. These use cases were then further refined through several customer meetings to ensure that all the requirements were captured. At the start of each mini deliverable, (see incremental) we then sat down and wrote specific cases for the one section.

This has proven to be very useful at gathering and displaying the requirements in a useful manner. Each use case forces us to look at the section and to determine all of the possible scenarios that we must account for. In most cases it provides us with a clear understanding of what our code is expected to do. We have found that although we had a use case outlining the functionality of one aspect of the program that this doesn't necessarily mean that you will implement it correctly. To the team the use case meant one thing but to our customer it described a completely different piece of functionality. Just because you have a use case written it doesn't mean that you understand it. We now know that we need to be more specific when we write the use cases. They need to be talked about more thoroughly to ensure that both sides understand the functionality associated with each case. We are currently preparing our second set of specific use cases that are associated with an increment. We will ensure that we understand all of the customer's expectations before we start the next phase.

2.4 Refactoring

One of the differences between the extreme programming approach and most typical project development approaches is the idea that a single piece of the overall project is designed and implemented (incremental deliverables) without any overall system design taken into consideration. This approach introduces a new idea called refactoring, whereby older pieces of code are redesigned to fit to the new pieces that are under development. The idea is that less time is spent designing the system and more time is spent actually building the system.

Taking this approach is an excellent way to reduce the time spent in system design, which is typically not adding any value to the final deliverable. The programmers also enjoy it because they get to the implementation portion of the process much quicker, and this is typically the part that designers enjoy the most. However, the process of refactoring should not become the dominant portion of the work being done on the project.

We found that initially the extreme approach required very little refactoring. Code was being developed quickly with few defects, working releases were available within a month and the team was experiencing success. The problems with the approach did not appear until later in the development of the project when there was a substantial amount of code present. New pieces of the project became more difficult to integrate because more code has to be considered when the additions were made. As well, more time was spent adjusting the existing code to make the new pieces fit into the system. It was also more difficult to nail down the customer requirements because only small portions of the system were considered in the planning stages of the project. This limited view of requirements capture also leads to a lot of refactoring because incorrect implementations needed to be changed.

In the end the majority of the programming time was spent in the refactoring process so that new features (that were still part of the deliverables) could be added to the system. We feel that more time was needed in the initial stages to plan out the design of the whole system so that these refactoring changes would not have been necessary. This is already a known problem in system design and most project managers know that more planning in the initial stages of development is absolutely necessary and reduces the time and cost of development. The extreme approach to project planning needs to consider some overall system design so that the refactoring process can be kept to a minimum. Some of the team members feel that refactoring is just repeated work and is not adding value to the end product.

2.5 Defects

The other nice part of the extreme process is the ability to catch defects fast. Smaller parts of the code are implemented so a person has a much better understanding of the section that is currently in development. This leads to less information slipping through the cracks and less defects. As well, deliverables are in very short duration so the customer sees the product as it is being developed and gives feedback instantly. This leads to many errors not actually becoming a defect because they do not reach the final product. We have had a positive experience with few defects in our code.

In our experience we had a total of three deliverables and only three defects. In fact, two of the defects that we found are related to incorrect requirements capture. The last defect required no rework because the refactoring that

was necessary for a missed requirement changed the algorithm where the defect was present and it simply disappeared.

2.6 *Unit Testing*

JUNIT is a unit testing software class that gives Java programmers the ability to test any portion of code they write. This involves defining a set of conditions that a test must pass as well as a testing set that the test belongs to. This is an integral part of the methodology of Extreme Programming (XP).

JUNIT allows for a hierarchical organization of the test cases, allowing the programmers to test localized classes, entire packages or the entire project. This type of environment is a necessity when refactoring the code, to ensure that minor changes do not propagate errors through the code.

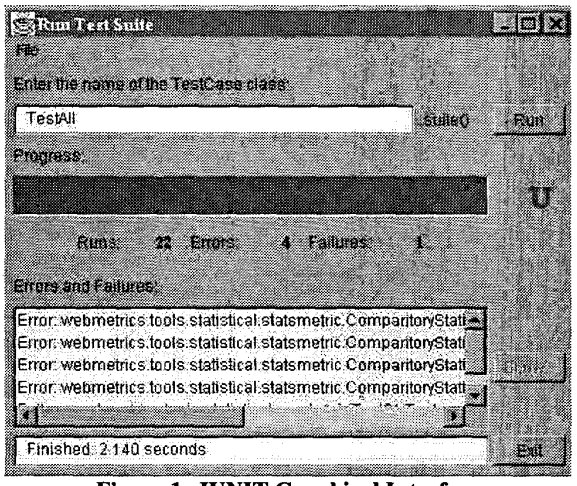

Figure 1: JUNIT Graphical Interface

Our group experienced and is still experiencing many hiccups with our use of JUNIT as an effective testing structure. During our development/design phase we originally neglected the unit testing aspect of XP. Certainly this has led to difficulties, as in any situation where requirements are not looked at in the beginning. The problem stemming from our ability to simultaneously develop code, while also trying to add on a testing structure to the entire piece of software has led to delays in progress.

Development of testing code is certainly progress, however, in light of the deadlines of our project, it at times feels like it is holding the team back. We recognize the usefulness of JUNIT as it has been implemented in a portion of our code and is continuously being added to.

The process of developing test cases prior to the implementation of the software is also an important part of the testing process of XP. The team has not made use of this process yet. XP suggests that the test and results are already known and the test is developed prior to the coding of the actual algorithm. With the limited number of bugs associated with our code, our lack of unit tests developed in advance does not present itself as a problem. In cases where the risk of error is much higher, this process would prove to be more beneficial

2.7 *Configuration Management*

Configuration Management is more than a beneficial process to a developing team of more than a single person; It allows for parallel development and reduces the risk of overwriting code.

Initially, our project group was involved with a very simple method of configuration management. This method involved whichever email had the most recent date was the most recent code. It also involved sending project members an email to state when a group member started and stopped developing the code. Upon completion of an addition to the code it would be emailed to the rest of the members.

As can be seen this method has many flaws. It allows for the possibility of overwriting of code and that concurrent programming will cancel out work done by one member, as it is hard to amalgamate two separate versions of code. With the small number of group members and the nature of the code that was developed the risk of overwriting code was reduced. Most partitions of the code were written initially as stand alone portions (modules) and then integrated at a later stage.

Recognizing that this was getting progressively more difficult, and also upon a recommendation by our project advisor, a move towards a more stable Configuration Management process was initiated. Using the Java Concurrent Versions System (JCVS) tool [3] has lead to a more consistent and relaxed method of coding. By containing all code on one server and accessing the server for the most recent version all programmers can be confident that they have the most recent alpha release of the software.

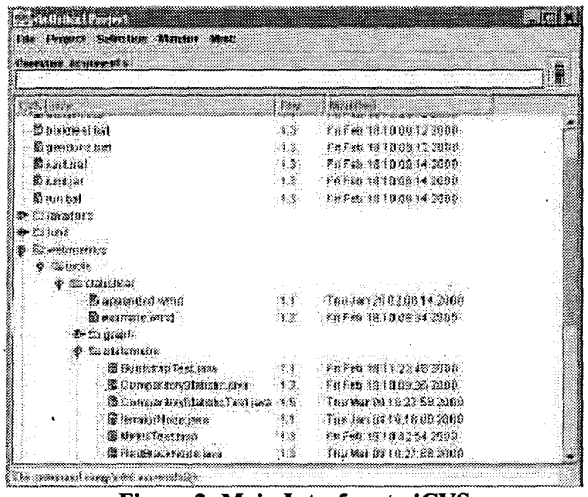

Figure 2: Main Interface to jCVS

JCVS has provided a much easier, much more reliable method of version control for the software. This method can certainly increase productivity and allow for parallel development. We encountered very few problems with the use of this tool and it has given us a much better sense of what state our code is in. This software also gives us the ability to comment on the reasoning behind changes being made and be able to track the software over the course of development much easier.

3 Conclusions

The common theme throughout this discussion has been the team's actual experience while implementing the Extreme Programming methodology. The XP paradigm presents an ideal framework within which to complete a software project. However, the process of implementing the ideal framework within a real project can cause difficulty.

In the university environment, the team found that paired programming hampered progress. Scheduling and the nature of the project were cited as reasons for why this concept didn't work.

Incremental deliverables are very important, because the customer always has a fully functional version of the software. Furthermore, the short revision times aid in project tracking and scheduling. Although the team firmly believes in the incremental requirements capture and deliverables, the project was still plagued by incorrect or missed requirements. Another problem, in the team's experience, with the incremental development was that of refactoring. Lack of planning and initial design is now being attributed to an overabundance of code rework at each new deliverable. The lack of software development experience has also contributed to this problem.

Unit testing and configuration management are both extremely useful in their own right. The team has been quite pleased with the implementation of these two aspects of Extreme Programming. Nevertheless, these aspects need to be implemented right from the start.

Again, there are several factors which contribute to the success of a software project, and that isn't just Extreme Programming. The team members need to have experience in software development, project management and the extreme programming paradigm.

At the end of the day, the customer is reasonably satisfied with the software product and the defect rate is extremely low. Even so we may have had difficulties implementing the XP process, we have still succeeded at two of the more important things. With more experience under our belt, the next iteration of our process will be that much better.

4 References

Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Don Mills, Ontario, 2000.

Fowler M., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, Don Mills, Ontario, 1999.

JCVS website, http://www.jcvs.org

Jeffries, R.E., et al, XProgramming website, http://www.xprogramming.com , 1999.

JUNIT website, http://www.xprogramming.com/software.htm

Meyers, R.H and R. E. Walpole, *Probability and Statistics for Engineers and Scientists*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.