XP as a Framework for Practical Software Engineering Experiments

Ralph Johan Back, Luka Milovanov, Ivan Porres and Viorel Preoteasa

TUCS Turku Centre for Computer Science and
Department of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A
FIN-20520 Turku, Finland

Abstract

We discuss how Extreme Programming (XP) can be used as the base software development method to perform practical experiments in software engineering. We show how the main features of XP can help us to minimize some of the problems and difficulties that appear when trying to perform such experiments in a university environment. We also discuss the execution and experiences from one experiment studying a new methodology: the Stepwise Feature Introduction.

Keywords

Extreme Programming, Software Engineering Research.

1 INTRODUCTION

The Software Engineering discipline studies how to build large software systems that fulfill the user's requirements, are reliable and are constructed on time and budget. This includes the study of many different concepts and techniques used in software development: software process models, modeling notations, programming languages and methods, testing and validation strategies, CASE tools, etc.

One of the problems that hinders the research and improvement of these techniques is the difficulty to perform significant controlled experiments. Many methods, such as Extreme Programming (XP) [4] or the Unified Modeling Language (UML) [16], have been conceived in the context of large industrial projects. However, in most cases, it is almost impossible to perform controlled experiments in an industrial setting. A company can rarely afford to develop the same product twice by the same team but using different methods, and then compare the resulting products and the team performances.

On the other hand, universities employ highly qualified research personnel that can employ considerable time to study better ways to build software, without the pressure of having to release new software products to the market. In this sense, a university setting could be the ideal place to perform practical experiments and test new ideas in software engineering.

However, researchers also find difficulties while testing new ideas in practice. First, it is possible that an experiment does not reflect the conditions found in a development company since researchers do not need to develop actual products. Secondly, university experiments must usually be performed by students. Students are not necessarily less capable than employed software developers, but they must be trained and their programming experience and motivation in the project may vary. There is also a high turnover rate as the students graduate and quit. Finally, although there is no market pressure, a researcher does not have unlimited funds, so it is necessary to optimize the costs of the experiments.

In this paper, we discuss how Extreme Programming can be applied as the base software process to perform practical experiments in software engineering in a university context. We think that many of the XP characteristic features help us to circumvent some of the problems described above. We also discuss the execution of an experiment using XP.

In this experiment we employed six undergraduate students under three summer months to develop an advanced text editor. We used XP as the base software process and we tested a new programming methodology currently under development: the Stepwise Feature Introduction (SWFI) [1].

2 XP AS A FRAMEWORK FOR EXPERIMENTS

In order to perform a non-trivial experiment on software engineering, we need to develop a non-trivial software application, or, at least, to simulate its development. Therefore, when planning an experiment we need to define what software must be developed and also which processes, methods, programming languages and tools will be used to build the software. These can be either well-established practices, such as C++ programming language and water-fall process model, or they can be new techniques that we want to try out.

The actual topics for an experiment are these new techniques and their impact on practical software development. To simplify the variability of an experiment and maximize its viability, it is better to test only one or two new techniques at a time. The question is which existing programming languages, software processes and tools are best suited as a base framework to perform these experiments.

In this section we outline the main characteristics behind the XP process and explain why they fit into our framework for experimenting in software engineering. We also propose some additional roles to the XP project staff which allow us to make the framework more effective.

2.1 Simplicity

One of the features that we appreciate most in XP is its simplicity. First of all, XP is easy to learn. Students learn it quickly, and they can learn it while doing what they like: programming. That does not imply that XP is easy to teach. The coach has a high responsibility for the education of the programmers and the adoption of XP by the team depends on the skills of the coach. Secondly, with XP it is possible to start a project in a short time.

In our case we had our first project running in two weeks, counting from the first meeting, where the members of the team where introduced, to the first integration of code. We were really satisfied with this time interval since all the developers where second year students and they were not familiar with XP.

2.2 Pair Programming and Team Dynamics

Pair programming has many significant benefits: better detailed design (in XP the design is performed on the fly), shorter program code and better communication within team members. Also, many common programming mistakes are caught as they are being typed, etc [7]. As it has been frequently reported [7, 8, 10, 15, 18], pair programming also has a great educational aspect. Programmers learn from each other while working in pairs. This is especially interesting in our context since in the same project we can have students with different programming experience.

Thanks to pair programming, we can expect that the senior students will teach the juniors while programming. Therefore programmer's training is more efficient since the learning continues as long as one programmer in a pair knows something that his partner does not. The two are so engaged in the coding task that it seems that much of the communication is non-verbal [13].

It seems that XP works better with a small number of developers. This is not a problem since in our experiments we can employ just a small number of programmers due to economical constrains. This helps to establish better communication within the team. The work of a coach is also more efficient in small groups.

2.3 Iteration Planning and On-site Customer

The on-site customer forces the development team to focus on the product, not on the experiment. In our case the customer can be another fellow researcher, who otherwise would not be involved in the experiment. Nothing, however, prevents the customer from participating in the project as a coach if he or she has the required skills. Short release cycles maximize feedback from the customer and force the team to have a working system as soon as possible.

We can use the XP iteration planning also to decide when to introduce or remove a method under test. The team should start the development of a product in an experiment using only proven techniques to learn the XP method thoroughly. Once the team has produced one or two iterations, it is time to introduce the experimental methods to test.

2.4 Collective Code Ownership

The concept of collective code ownership is also necessary since we expect a high turnover of students between different experiments, many of which may be consecutive stages in a long range product development effort. New programmers need to read and understand the code that was written before, so the code should be very readable and understandable. As described in [17], a well-defined coding standard and the sharing culture encouraged by pair programming can help in this respect

Another important issue is the legal ownership of the code. According to Finnish law, code developed by an employed programmer (including a student employed as a programmer) is owned by his employer. However, the code developed by a student as part of an exercise is own by the student.

To avoid any possible legal issue in this matter we decided to use an open source license for the code produced in our experiments. This does not actually solve the problem of who owns the code, but it ensures that the code will always be available for inspection, modification and publication.

2.5 Continuous Integration

Continuous integration has an interesting side effect that we discovered when analyzing the results of the project. The repository used for integration contains an invaluable trace of the activity of the programmers. For example, CVS [5], an open source version control system, keeps track of who performs each check-in and when. This information can be retrieved and analyzed later on, as it was done in [11]. This helps us to monitor and measure the speed of the development in an easy and unobtrusive way: the programmers do not even need to know how their work is monitored. We consider this ethical, since the programmers know from the very beginning that they participate in an experiment and that their work will be monitored.

2.6 Additional Roles in Experimental Projects

Based on the facts mentioned above, we found that XP is well suited as a framework for practical software engineering experiments. However, besides the standard roles in XP project such as programmer, customer and coach, we need to introduce some additional roles: lecturer and methodologist.

The task of a lecturer is to give short tutorials to the programmers and perhaps even coach at the beginning of the project. The goal of a tutorial is to give an overall introduction of the concepts or techniques that will be used in the project. Tutorials should not teach everything, they just give the strictly necessary information for starting the project. Tutorials are necessary, since our programmers may not have all appropriate knowledge and working experience for the project. For example, a full course on design patterns [9] can be 30 hours long. In about two to four hours it is, however possible to introduce the idea of patterns and teach one or two patterns that the coach thinks will be used in the project.

Ideally, a lecturer appears only at the very beginning of

the project. However, he should be available for consultations during the length of the project to explain issues that the coaches cannot resolve.

The task of a methodologist is to ensure that the methods under test are applied correctly. Unlike a lecturer, a methodologist has to participate in the project throughout its whole life, but unlike the coach, he does not need to be in close contact with the team on a daily basis.

It is not necessary that different people play the roles of lecture and methodologist. Having lecturers involved in a project as coaches would be useful because they know the concepts that are used very well. However, the work of a methodologist requires a lot of effort and can be more efficient if he or she does not have other roles in the project.

3 USING XP IN AN ACTUAL EXPERIMENT

In this section, we describe our first complete experiment sing XP as the base software process. The objective of the experiment was to test the methods of Stepwise Feature Introduction in practice. Since it was our first attempt to use XP in an experiment, we were also testing how XP performs in a university setting. A further goal was to see whether a non-trivial piece of software can be produced by inexperienced students in a short time span. The product to be built was an outline text editor [2].

The project was carried out by ten persons. One professor acted as customer and methodologist. Three Ph.D. students acted as lecturers and coaches. Finally, six undergraduate students were the developers. Most of the students had completed their second year in the computer science/engineering curriculum and had basic courses in programming but not all of them had courses on software engineering. None of the students were familiar with XP or SWFI.

The programming language of the project was Python [12]. To keep track of the project assets, we used the CVS version control system. One of the students had some experience with Python before and none of them was familiar with CVS.

3.1 Stepwise Feature Introduction

Stepwise Feature Introduction is a software development methodology based on incremental extensions of an object-oriented software system with only one new feature at a time.

It has much in common with the original stepwise refinement method [3], the main difference being that software is built bottom-up with an emphasis on object-oriented programming.

According to this methodology, a software system should be built in thin layers where each successive layer introduces a new feature to the system and does not break the functionality implemented in previous layers. Each layer represents a running system with a functionality that extends the previous layer. The basic layer is a system with minimal functionality.

Each successive layer is described as a collection of classes that are extensions of classes from the previous layer. The extension is implemented using inheritance (also using multiple inheritance), delegation or forwarding. The inherited methods are supposed to remain unmodified or to be redefined such that the new code achieves the same effect as the inherited one, plus possibly some additional effect on newly introduced attributes. All class members, once introduced, should be present in the extensions of a class through all of the higher layers.

3.2 Organization of the Project

The project ran during three summer months in three phases: training, programming and cleaning up. The project started with a meeting where all the members were introduced to each other. The overall working conditions and the schedule were discussed.

The first phase of the project took the first two weeks. The students attended tutorials on XP, Python and one of its user interface toolkits, design patterns, and CVS. They were also introduced to the product and its requirements. The average time spent on each tutorial was three hours.

The second phase was the main and the longest and it took nine weeks. During this time the project was carried out according to the guidelines that were presented to the students in the learning phase. The PhD students were coaching and helping the students with the tasks. When the overall structure of the code started to become too complicated the coaches got more involved to help the students to refactor and simplify the code.

In the last phase we stopped introducing functionality to the editor and focused on debugging, code cleaning and on reviewing and writing documentation. There was also some work to increase the performance of the product.

3.3 Experiences and Impressions

We managed to build the project on time and, what is more important, we managed to extract a lot of useful feedback about the method [2].

The learning part of the project turned out to be very useful. As it was also observed in [6], well-chosen short tutorials focused on topics that are needed in the project (techniques, tools) got the project running in no time.

Pair programming worked well in our project. At the beginning we mixed the students who knew Python with the ones that were learning it. This fact leads to a quick start. However, pair programming did not work so well during debugging. The students complained that it was easier and faster to debug the code alone since "everybody has a different theory about where the bug is".

Our programmers did not like to write tests before the actual code. They considered it counterintuitive and preferred to write the tests while or after the code.

At the beginning of the project we also wanted to use the design by contracts [14] technique, but it turned out that we wanted too much in too short time. In the future

we will refrain ourselves from introducing too many new concepts at the same time to inexperienced students.

4 CONCLUSIONS

Experiments in software engineering are expensive and difficult to setup. Extreme Programming can help us to study new software construction methods by providing a flexible software process that is easy to learn, keeps the programmers focused on the product and not on the experiment and allows us to observe the results of the programmers right from the beginning. We have shown that using XP we can perform such experiments in a university setting and the experiments are performed faster and with less effort [2].

However, we cannot always apply XP as described in [4]. There are some cases where we definitely cannot apply XP in an experiment, e.g. if the topic of the experiment is another software process. In other cases, we can use XP, but we still need to adapt it so that some task is emphasized more often than it usually is. For example, if the topic of an experiment is a modeling notation such as UML, we need to be sure that modeling appears as a central task in the software process used in the experiment.

ACKNOWLEDGMENTS

We would like to thank Rasmus Back, Linus Bernas, Tomas Czarnecki, Marcus Eriksson, Mats Sjöberg and Max Söderström for their participation in the experiment described in this paper.

REFERENCES

[1] R. J. Back. Software Construction by Stepwise Feature Introduction. To appear in Proceedings of the ZB2001- Second International Z and B Conference, Springer Verlag LNCS Series, 2002.

[2] R. J. Back, L. Milovanov, I. Porres and V. Preoteasa. An Experiment on Extreme Programming and Stepwise Feature Introduction. Technical Report No 451. Turku Centre for Computer Science. March 2002.

[3] R. J. Back and J. von Wright. Refinement Calculus -A Systematic Introduction. Springer-Verlag, 1998.

[4] K. Beck. Extreme Programming Explained: Embrace hange. Addison-Wesley, 1999.

[5] B. Berliner. CVS Web Site. Available electronically at: htp://www.cvshome.org/.

[6] K. Boutin. Introducing Extreme Programming in a Research and Development Laboratory. In G. Succi and M. Marchesi, editors, Extreme Programming Examined. Addison-Wesley, 2001.

[7] A. Cockburn and L. Williams. The Costs and Benefits of Pair Programming. In Proceedings of eXtreme Programming and Flexible Processes in Software Engineering XP2000, 2000.

[8] L. L. Constantine. Constantine on Peopleware. Englewood Cliffs: Prentice Hall, 1995.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison-Wesley, 1995.

[10] D. H. Johnson and J. Caristi. Extreme Programming and the Software Design Course. In Proceedings of XP Universe, 2001.

[11] S. Koch and G. Schneider. Results from Software Engineering Research into Open Source Development Projects Using Public Data. 2000. Available at: http://opensource.mit.edu/papers/kochossoftwareengineering.pdf.

[12] M. Lutz. Programming Python. O'Reily, 1996.

[13] R. C. Martin. RUP / XP Guidelines: Pair Programming. ational Software White Paper. Available at:http://www.cs.unb.ca/profs/wdu/cs4025/pairprogramming.pdf, 2000.

[14] B. Meyer. Object-Oriented Software Construction. Prentice Hall, second edition edition, 1997.

[15] M. M. Müller and W. F. Tichy. Case Study: Extreme Programming in a University Environment. In Proceedings of the 23rd Conference on Software Engineering. IEEE Computer Society, 2001.

[16] OMG. OMG Unified Language Specification. Version 1.4, February 2001, available at: http://www.omg.org.

[17] W. C. Wake. Extreme Programming Explored. The XP series. Addison-Wesley, 2000.

[18] L. A. Williams and R. R. Kessler. Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom. Journal on Software Engineering Education, December 2000.