A Formal Experiment Comparing Extreme Programming with Traditional Software Construction

Francisco Macias, Mike Holcombe, Marian Gheorghe
University of Sheffield
Regent Court
211 Portobello Street
Sheffield, S1 4DP, UK
+44 114 222 1800
{f.macias, m.holcombe, m.gheorghe}@dcs.shef.ac.uk

Abstract

The paper describes an experiment carried out during the Spring/2002 academic semester with computer science students at the University of Sheffield. The aim of the experiment was to assess extreme programming and compare it with a traditional approach. With this purpose the students constructed software for real clients. We observed 20 teams working for 4 clients. Ten teams worked with extreme programming and ten with the traditional approach. In terms of quality and size teams working with extreme programming produced similar final products to traditional teams. The major implication for the current practice of traditional software engineering is that in spite of the absence of design and the presence of testing before coding the product obtained still has similar quality and size. The implication for extreme programming is the possibility of growth and maturation given the fact that it provided results that were as good as those from the traditional approach.

Keywords: empirical software engineering, extreme programming, formal experiment, Software Hut Project.

1. Introduction

Students of Computer Science usually attend practical modules devoted to integrating the knowledge that they have acquired previously during their course. Very often in such modules the students build a software system in teams. The undergraduate students at the University of Sheffield attend one of these integrator modules during the 4th semester of their course. The name of this module is the Software Hut Project. For this practical module, the University contacts potential clients and the students produce software for them. This module provides an opportunity to carry out formal experiments given that the environment looks more like that of industry but has the advantages of providing an opportunity for an *in vitro* environment. This condition makes Software Hut Project a good candidate for the assessment of a software system production process. The name of the process we are interested in is extreme programming. Extreme programming is a new philosophy that encourages a number of practices including team communication. It encourages the extensive use of testing and favours simplicity instead of complex solutions. Extreme programming simplifies the traditional approach, filters the old practices and rearranges these practices.

The aim of the research is to assess extreme programming through a formal experiment. In order to do that, a comparison between extreme programming and traditional programming took place. The

students doing computer science at the University of Sheffield produce bespoke software for their clients in teams; half of the student teams used extreme programming and the other half used a traditional approach. The students were closely observed during the academic semester and their work was measured. This report describes the method followed and the findings of the experiment. This experiment is relevant as it provides an assessment of the whole software development process rather than a part of it.

Section 2 includes a brief description of extreme programming and the traditional approach. Section 3 depicts the environment of the experiment, which includes the skills and expertise of the students as well as the description of the projects they were producing during the academic semester. The objective and the hypotheses are also introduced. In the same section the formal definition of the experiment is given, which includes the metrics and the collection method. Section 4 includes a brief description of the raw data while Section 5 presents the analysis and interpretation of these data.

2. Background

This experiment compares two treatments: traditional and extreme programming. Traditional programming or the traditional approach is a process that includes the steps of the classical waterfall model before delivering the software. These steps are: Analysis, Design, Implementation and Testing. It does not matter whether these steps follow the Object Oriented or structured methodologies.

Extreme programming (XP) is a discipline of software development. It emphasises productivity, flexibility, informality, teamwork, and the limited use of technology outside of programming. Rather than planning, analysing, and designing for the distant future, XP programmers do all of these activities -a little at a time- throughout development [Beck 98].

One important idea behind extreme programming (XP) is working in short cycles. In XP the basic unit of time is the cycle. Every cycle starts by choosing a subset of requirements (stories) from a larger, more complete set. The client must be engaged in this selection process. Once such a subset has been selected, the functional test sets for every requirement must be written. The members of the team then write the code, working in pairs. Each piece of the code is tested according to the defined functional tests. The client must be engaged in this activity, also each cycle takes between one and four weeks to complete. The first iteration puts the metaphor of the architecture in place. There is no architecture, instead there are one or several metaphors.

XP relies on four values; simplicity, communication, testing and courage, and each practice enhances the others. Simplicity stands for a way of representation and construction that always starts from the simplest task and maintains the items and the structure of the body in such simple conditions. Communication should avoid the use of technology and promote face to face communication. It also encourages the exchange of ideas among the people engaged in the project: developers, clients and managers. Testing must be carried out at all levels, but more importantly, testing drives the implementation because the functional tests for every piece of code should be ready before starting to write such a code. Courage refers to the self-confidence in which the members of the team must have to address problems. If a new challenge requires a new kind of solution, it is important to try it, or if some work already completed goes wrong, it is important not

to hesitate to throw it away; start over again, instead of trying to fix or recover it. Extreme programming also addresses the changing nature of requirements, in fact it encourages it. The idea behind these values is to define the shape of this philosophy as human oriented [Beck 99-2].

XP involves the use of 12 practices [Beck 99-1]. These are: Planning game, Small releases, Metaphor, Simple design, Test, Refactoring, Pair programming, Continuous integration, Collective ownership, On-site customer, 40-hour weeks, and Coding standards. "Planning game" stands for small cycles. This means that the client will select the stories (requirements), previously written by the team members, that should be implemented while the developers define the time that will be spent in this cycle. "Small releases" refers to these small products delivered at the end of every cycle, which will be integrated later into the rest of the system. "Metaphor" speaks about the ideas depicted through metaphors that represent the architecture of the system. Simple design means that the design should be the simplest possible that provides business value. "Test" refers to testing all the products and sub-products derived from the process continuously. "Refactoring" refers to the modification of the code in order to leave it ready for a further extension and maintenance. "Pair programming" means that the code should be produced working in pairs: two programmers, one screen, one mouse and one keyboard. "Continuous integration" speaks about the growth of the system through the integration of small releases that are delivered in every cycle. "Collective ownership" stands for the capacity that all the members of the team have to modify or extend any part of the system. "On-site" customer refers to the fact that the client should be present during the production process. "Forty-hour weeks" states that quantity of work must be balanced with quantity of rest in order to maintain quality. "Coding standards" means that the members of the team must follow the rules stated (by the members of the team) for the production process.

3. Methodology

The objective of the experiment was to provide scientific evidence to either give support or to reject the claim that extreme programming is a valid software production process, which is better than the traditional approach.

Before running the experiment, a pilot study was carried out [Holcombe 01, Macias 02]. This showed that it would be possible to point out some of the aspects that enhance extreme programming and those aspects that weaken it. After the pilot study, the original hypothesis remained in its original shape. The pilot study ran during the spring of 2001 while the experiment ran during the spring academic semester of 2002, both of them at the University of Sheffield. This hypothesis is presented in the next subsection and the further subsection presents the design of the experiment and presents the metrics as well.

3.1 Experimental context

The people engaged in the Software Hut Project (SHP) were 4th undergraduate semester students. This fact represented an advantage since all of them had roughly the same expertise and they had acquired similar skills. When the students reach the 4th semester, they already know how to write a program, how to produce data structures, to write a specification, to produce a web page and they have acquired organisational skills. The Software Hut Project is a module useful for "integration and developing of skills". The 93 students registered in this module (SHP) attended one course of training during the semester in one of two topics: extreme programming or the traditional design-led process for software construction. For this purpose, the group of students was divided in two

halves. The training was provided during the normal sessions of the module. At the same time as these sessions, the students had to interview their clients and plan their project.

There were four external clients. These clients had different requirements, and these were judged by the lecturer to be feasible within the time and other constrains. The clients (so-called A, B, C and D) were as follows:

Client A. The primary role of Small Firms Enterprise Development Initiative (SFEDI) is to increase the ability of the self-employed, owners and managers of small companies to start up, survive and thrive. The organisation provides advice and support to small businesses nationally. SFEDI wanted a web site for their employees that would let them distribute general documents, policies and procedures to other employees. They wanted to make them accessible away from their main office. They wanted to restrict the access to certain documents, according to the category of the employee accessing them. Documents contained within the system fall into two categories: those that need only to be read (non-interactive documents) and those that need to be filled in (interactive documents). Their main aim was to improve internal communications among employees. SFEDI employees have a fairly good level of computer literacy.

Client B. The School of Clinical Dentistry of the University of Sheffield conducts research using questionnaires to collect information about patients. They may run several questionnaires simultaneously. The data generated from these questionnaires is used for a variety of purposes.
The school required a system that allows them to customise the on-line questionnaires and subsequently produce a file containing the data submitted. Security was a primary concern. Every questionnaire should have its own password. A person asked to fill in a certain form receives a password in order to access the questionnaire. Additionally they should remain secure when it is transferred from the client machine to the database. The generation of the questionnaire should be very simple and will not require any specialised knowledge; as such it will be usable by anyone with low computer literacy.

Client C. University for Industry (UFI) was created as the government's flagship for lifelong learning. It was created with its partner Learn Direct, the largest publicly founded on-line learning service in the UK. This initiative encourages adult education. In order to analyse performance and predict future trends UFI Learn Direct needs to collate and analyse information such as the number of web-site hits or help-line calls at different times of the day and the number of new registrations. They also need to know how these items of data relate to each other.

The proposed problem was to design a statistical analysis programme for UFI. The systems' main use would be to help managers at UFI plan how best to allocate and manage their resources based on trends and patterns in the data recorded. The proposed system will take the collected data as input in the form of a comma-separated-values file. Data are recorded on the following aspects: concurrency, year trend, performance indicators, users (hits) per hour and predicted growth. It then constructs relationships between the different variables. This information is then processed and returned in graphical form. The system will have two types of users interacting with it. The first one is the Main System user, s/he is technically competent and capable of understanding quite complex user interfaces. The second one is the intranet user. The range of ability among these users of the intranet is wide.

Client D. The National Health Service (NHS) Cancer Screening Program keeps an archive of journals and key articles that they provide to the Department of Health, the public, the media and people invited for screening. They required a system which was simple to use and easy to maintain which allowed them to:
- Catalogue the existing collection of articles,
- Add new articles,
- Expand the collection to include articles on screening for colorectal, prostate and maybe other cancers,
- Link associated articles,
- Find and retrieve articles quickly and effectively.

A member of the staff will maintain the system, but other staff members will use it to search for articles. Thus, users are the staff of the NHS Cancer Screening Program National Co-ordination Team. As such they have mixed IT skills. All are capable of operating self-evident systems such as commercial word processors and web browsers, but not a program that requires more specific knowledge or extensive training. They therefore require a system which is simple to install, maintain and of course use, so the client has no special preference for the system appearance or operation beyond the requirement that it should be easy to use.

The elicitation of the requirements was a major and lengthy process of negotiation. The formal requirement documents were agreed between the teams and their clients. The completed systems had to be installed and commissioned at the clients' site. Finally the client should give a mark to the software by assessing several external quality elements while the lecturer should give a mark by assessing internal quality elements.

Every team of students worked with only one client. Some students wrote programs in Java, while others wrote code for PHP and SQL according to the requirements of the system. From the objectives pointed out in this subsection and their connection with the background theory, the following hypotheses were stated:

Null Hypothesis: The use of extreme programming in a software construction process produces as good results (external and internal quality) as those obtained from the use of traditional processes. (in the average case of a small/medium size project)

Alternative Hypothesis: The use of extreme programming in a software construction process produces different results than those obtained from the application of traditional processes.

These hypotheses emerge from the discussion about the validity of extreme programming. There are managers and developers who see extreme programming as a suitable alternative for software process production [Moore 02, Putnam 02, Rumpe 02, Wright 02]. Extreme programming (as previously pointed out) has been called extreme because of the high risk derived from the use of low level technology and the separation of the traditional establishment for software production processes such as detailed design stages. Some of the major diversions posed by extreme programming are the reduced, or null, presence of design and the construction process based on black box testing (see details in section 2).

3.2 Experimental design

The population sampled includes all the teams engaged in the Software Hut Project (see details in section 3.1). Some possible bias could be present when some students have had practical experience outside the University but this possibility is unusual. In the SHP 2002 module there were no such cases.

There are several printed guidelines that help to organise the experiment [Kitchenham 01, Shepperd 95]. The organisation of the experiment corresponds to a Randomised Complete Block Design, so the experimental units were randomly allocated and every block received the two treatments. There were two treatments: extreme programming and traditional approach. Formerly the students gathered in teams. Then they received a notification of the treatment (extreme or traditional) and client (A, B, C or D) they had to work with. The lecturers distributed randomly the teams, among the clients and treatments. There were 20 teams and four clients. Then every client received both treatments (extreme and traditional), and five teams were allocated to each client. Each team tried to provide a complete solution for their client. This means that two clients had two teams working with extreme programming and three teams working with traditional programming, while the other two clients had three teams working with extreme programming and two teams working with traditional approach (Table 1). Treatments were defined in section 2.

Blocks		Treatments: XP	Treatments: Traditional
	A	5, 7, 8	18, 20
	B	2, 6	12, 14, 17
	C	1, 9	11, 13, 19
	D	3, 4, 10	15, 16

Table 1. Distribution of teams per treatment and blocks (clients)

There were 20 experimental units. The 93 students of the Software Hut Project gathered in 20 teams. For the purpose of the study they were never observed or tracked individually, but always as teams. The communication processes, assessment, log and verification were always at team level. Then the unit of observation was the team and the experimental unit was the team as well. Every experimental unit, that is every team, dealt with only one treatment and only one client (allocation block).

The size of every team usually was 5 but some teams only had 4 people. Given the small amount of experimental units per treatment (10) all the teams engaged with the Software Hut Project were included in the experiment. There was no blindness among the teams (students) as they received suitable training to apply the treatment. The lecturers and the clients were also fully aware of the different treatments each team was working with. The lecturers provided the training for the treatment. The clients could identify the extreme programming teams because the treatment encourages a close relationship with the client.

There were three main factors to measure: Time spent in the production process, Quality of the product and Size of the product. The students reported the time that they spent in the project. Every team submitted weekly timesheets that included the distribution per every member of the team in

every activity. The quality of the product was divided in two aspects: external and internal. External quality aspects were assessed by the clients, whilst internal quality aspects were assessed by the lecturers. The size of the products was obtained from the reports of the teams. In order to choose the metrics, the GQM goal template [Basili 86, Basili 88] was used.

The factor that refers to time spent included seven different aspects. The external quality factors included 10 metrics. Internal quality factors included seven metrics in the extreme programming treatment while traditional programming treatment included six metrics. The size of the product included five metrics.

The metrics included for the purpose of measuring the time, were the number of hours spent by every member of the team every week in these activities: research, requirements, specification and design, coding, testing, reviewing and other activities.

The external quality aspects, assessed by the client, were divided into two aspects. The first aspect was documentation, which includes presentation, user manual and installation guide. The second was the software system, which includes ease of use, error handling, understandability (use of appropriate language), base functionality (completeness), innovation (extra features), robustness (correctness -does not crash), and happiness with product. Client D did not assess robustness given the nature of the required system (see section 3.1).

The Internal Quality aspects assessed were not exactly the same in both treatments. There were slight differences because it was necessary to tailor the metrics to the treatment (e.g. in a traditional approach it is not required to provide evidence of pair programming). In extreme programming the metrics were: requirements documentation (a set of user stories and a set of requirements signed off by the client), a detailed specification of test cases for the proposed system (using an appropriate language), the test management processes, the completed test results, the code, user documentation (installation instructions and maintenance guide), and a general description (including log) of the project. In the traditional approach the metrics were: requirements documentation, the detailed design for the system, completed test results, the code, users documentation (installation instruction and maintenance guide), and a general description, including the log of the project.

In order to assess the size of the product, five metrics were observed: the number of functional requirements, the number of non-functional requirements, the number of test cases, and the size of the code (number of logical units and number of lines of code).

The data required for the assessment was collected in two stages. The first stage was during the semester and the second was at the end of the project. The data collected during the semester was mainly presented in the form of minutes and timesheets. The meeting minutes contained the revision of the activities of the current week and the plan of the work for the next week. The timesheets registered the time spent in every activity during the current week (if it has ended, or the previous week if the current one has not ended). Both timesheets and minutes were automatically collected. A script ran every Friday by 4:00 and updated the records of the teams with the new files. The information was gathered in order to produce the report and look for inconsistencies, if any. This was registered with the purpose of verifying the information in a weekly meeting. In the second stage the clients and the lecturers were asked to provide their marking. This information

was collected by the end of the semester, and the reports of the teams were also reviewed by the end of the semester. The marking provided the information about the quality while the revision of the reports provided the information about the size of the project.

4. Results

The pilot study ran during spring 2001. It made us aware of the importance of teams following the treatments, extreme programming and traditional, properly. The only way to be sure of that, and apply the corrections if required, was to track the activities. For this purpose the timesheets and the weekly minutes were valuable. According to the time spent (information from timesheets) and the activities (information from minutes) extreme programming teams spent less time (of its total amount of time) in programming and more time in testing while people in traditional teams spent more time in programming and less time in testing. People working in the extreme programming approach spent much less time in Analysis and Design. Figures 1 and 2 present the distribution of the time spent. These diagrams present percentages. For the actual amounts of time see Appendix 1.

In general, the total amount of time spent in the project was higher in the extreme programming teams. Not only was the distribution of time among activities different between one treatment and the other but the moment that such activities started to appear during the semester also varied. This means that testing activities started to appear almost simultaneously with coding in extreme programming timesheets while in traditional team timesheets testing often appears after the coding; coding appears sooner in extreme programming compared to the traditional approach. Traditional teams spent much more time in analysis and design than extreme programming teams and these extreme programming teams produced very small, simple designs. In some cases they did not produce any at all. The distribution of time and the activities carried out shows that the teams seemed to follow the appropriate methodology for each approach in general terms. The average time spent by the five teams of every treatment was nearly the same as the overall mean.

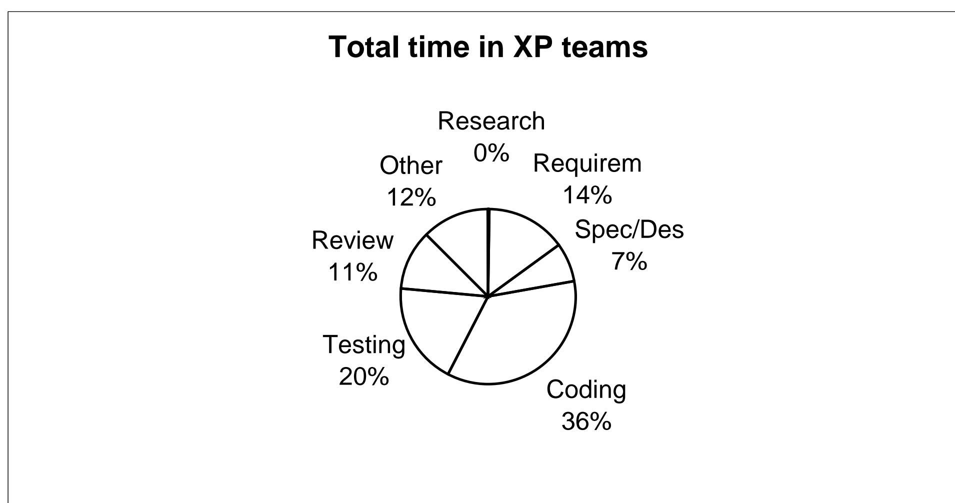

Figure 1. Distribution (from 100%) of time in extreme programming teams.

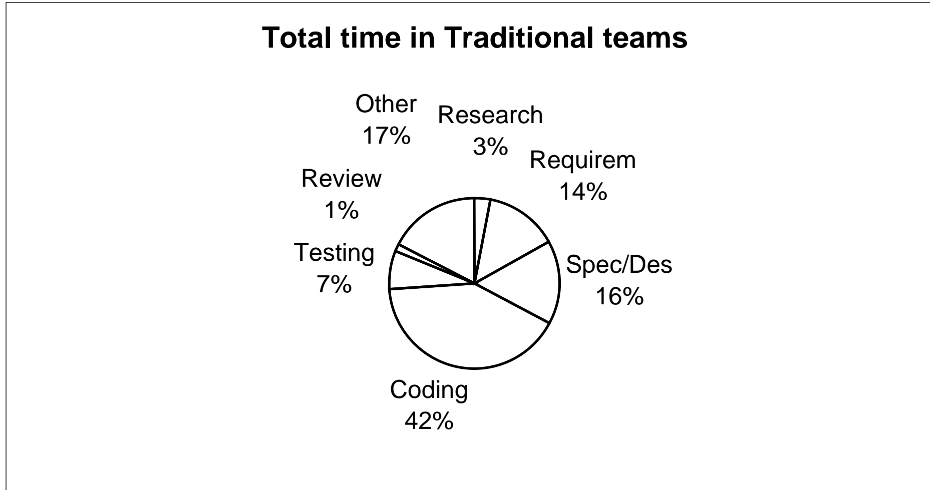

Figure 2. Distribution (from 100%) of time in traditional approach teams.

The distribution of activities during the time provided enough evidence to accept that the teams under extreme programming treatment followed it at an acceptable level. This result was important as it is the first requirement before going on with the experiment.

According to the design [Montgomery 01], the model for the experiment was:

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_k + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk} + \epsilon_{ijk} \qquad (\mathrm{I})$$

where y_{ijk} is the ijkth observation, μ is the overall mean, α_i is the ith principal effect of the treatment, β_j is the jth principal effect of the team, γ_k is the kth principal effect of the block, $(\alpha\beta)_{ij}$ is the ijth first order interaction between treatment and team, $(\alpha\gamma)_{ik}$ is the ikth first order interaction between treatment and block, $(\beta\gamma)_{jk}$ is the jkth first order interaction between team and block, $(\alpha\beta\gamma)_{ijk}$ is the ijkth second order interaction between treatment, team and block and ϵ_{ijk} is the random error component of the model. About the indices, 'i' is the treatment, extreme or traditional, 'j' is the team, from 1 to 20 and 'k' is the block, or client, A, B, C or D.

The Analysis of Variance (ANOVA) for the time, showed that this factor was ruled by the treatment but not by the block (see Table 2). The number of metrics observed using the timesheets was seven but for the ANOVA test only the total amount of time for all the activities during the complete process was considered. The confidence interval obtained from the F-test showed that teams under extreme programming expected more time than teams in traditional programming. The tool used to gauge the data of Table 2 was Minitab.

The factor quality had two different aspects: External and Internal. The clients ranked the External quality whilst Internal quality was ranked by the lecturers. There was no apparent relationship between both aspects, and the correlation coefficient between them was 0.33. For External quality, 10 items were measured in three of the four blocks and 9 items in one block (NHS Cancer Screening Programme, see Appendix 1). In Internal quality, 7 items were measured in the extreme programming treatment and 6 items in the traditional treatment. One quick look at the data of quality showed that the average for the extreme programming treatment was higher in all cases, that

means: External quality, Internal quality, as well as the sum of both, were between 3% and 6% higher. The two quality factors were considered for the ANOVA, first separately and then together. In other words, the ANOVA received three sets this time, the first one was the total sum of the items for Internal quality, the second was the total sum of the values for External quality, and the third was the total sum of both aspects. According to this test neither the treatment nor the block had any influence on the response, either separated or together (see Table 2).

In order to assess the size, the total amount of test cases and the total amount of requirements were considered. In addition to the test cases, the teams had to produce a document with the results of the test sets once applied. The number of test cases was counted and the total amount used to obtain the ANOVA for the treatment and the ANOVA for the block. In both cases the null hypothesis held. The requirements written by the teams were classified in two groups: functional requirements and non-functional requirements. The first group was sub-classified into high, medium and low priority, whilst the second group was sub-classified into five items. The total amount of requirements was F-tested against the treatment and the block. Again, the test did not provide evidence supporting the alternative hypothesis, neither for the treatment nor for the block (see Table 2).

	F	P
Time vs Treat	6.48	0.02
Time vs Block	0.08	0.97
Qual vs Treat	1.39	0.254
Qual vs Block	0.01	0.999
Ext. Q. vs Treat	0.65	0.431
Ext. Q vs Block	0.08	0.972
Int. Q vs Treat	1.87	0.188
Int. Q. vs Block	0.21	0.886
Test C vs Treat	1.53	0.232
Test C vs Block	1.21	0.339
Req vs Treat	0.02	0.891
Req vs Block	2.97	0.063

Table 2. Results of the Analysis of Variance (ANOVA)

In general terms, the only factor with observable dependence of the treatment was the time spent by the teams. The other factors (quality and size) appeared to have a similar behaviour no matter what treatment the team followed. Sometimes a factor could depend almost of the block e.g. requirements vs. block. So, apart from time, we can say that extreme programming and the traditional approach provide roughly the same results.

6. Discussion

The first challenge faced during the experiment was to ensure that teams followed the treatment. Hence the importance of the timesheets and minutes. From the results it was possible to establish that the teams generally followed their respective treatments. The teams dealing with the extreme programming treatments spent, on average, more time than the teams dealing with the traditional treatment. This fact is easy to explain as extreme programming encourages communication. A good

example of this is pair programming. Pair programming requires two people working simultaneously with the same piece of code. This way of working has many advantages, e.g. the quality of the code is higher [Williams 00], the skills of the members of the teams develop more evenly, and the success of the project does not rely on a super-programmer but on teamwork.

Minimising the stage of Design in a software construction process is in itself a revolutionary step. The traditional software construction process, including a well-defined and well-distinguished Design stage, is widely accepted, and in fact, some variants of the traditional process promote the production of a very finely defined Design. Design has two mainstreams: the architecture of the system and the details for further implementation. Extreme programming substitutes architecture with an overall metaphor, and the details for implementation with the implementation itself. The idea here is: if you have to think about and then write all these details, do it straight to code and avoid the intermediary step; this means do not write it twice [Beck 99-2, Fowler 01]. And then if the requirements change there are fewer overheads.

Extreme programming encourages simplicity, particularly in the Design. In this experiment we have seen that teams working with a Design-less production process obtain similar results (sometimes slightly better) than teams working with the traditional Design-led process. One would have expected that if you remove an important piece of construction (Design) from a process (software construction) you will not be able to obtain the same quality or complete product, but something rather strange, for example an incomplete or badly functioning product. This is an important result and points to the value of Simplicity and the practice of direct implementation of extreme programming.

An objective discussion should consider the uneven situation of the treatments, given the fact that one of them is more mature than the other. It is to be expected that a new procedure may lack maturity as a process. The development of the process removes unnecessary steps, emphasises relevant aspects and provides health and strength to the whole process. If it is expected that such a procedure has wide use, scope or impact the maturity process could be long and difficult. The traditional approach has been tested in many different situations, and the general frame of Analysis-Design-Implementation-Testing-Maintenance has been accepted even further than the scope of a software construction process. Extreme programming does not have such a privilege; it is a very new idea and is only in the initial stages. From this perspective, it is surprising that extreme programming has provided as good a result as the traditional approach. We are far from being able to predict which subset of the practices could survive. It could be all of them or only a few of them; maybe the practitioners will provide new practices. What we have seen is that teams working with black box testing-based analysis, testing based on requirements, simple design, planning game, pair programming, coding standards, collective ownership, continuous integration, small releases and minor scale of metaphors and refactoring have been as successful as teams working with a traditional approach which emphasised testing and standards. We are aware that no team worked either 40 hours week or with an on-site customer and so one could argue that the full extreme programming process was not used.

Internal quality factors were not related to external quality factors. External quality factors refer to those that can be detected by users and were based on the final products. Internal quality factors related to quality of process and intermediate deliverables and documents. The client assessed the

external, and the lecturers assessed the internal. Two clients, NHS Cancer Screening Program and the School of Dentistry asked for simple systems. The simpler the interface the happier the client. They required systems that were simple to learn, use, and maintain. The other two users did not look for simplicity but for more innovative systems. On the other hand, the lecturers assessed extreme programming teams and traditional teams in different aspects, according to the treatment, e.g. traditional teams were assessed on the detailed design while extreme programming teams were assessed on the specification of test cases. Neither external quality nor internal quality were always assessed under exactly the same rules for all the teams, but despite the differences the variability was low, as observed in the ANOVA results. Looking for a relationship among both quality factors, we ran a correlation coefficient test. It showed a very low (0.33) possibility of relationship among them.

There are some other factors that we can only infer, as they are not easy to measure. Among the aspects we have to consider there is the cost of the technology and the cost of the coaching required in order to maintain the practices. Technology always has a high cost; indeed, often the higher the technology the higher the cost. Extreme programming was originally thought of as a low technology requiring process [Beck 98]. Even with other added characteristics, extreme programming still remains a low technology requiring process. Such a requirement makes it less expensive than other approaches that require expensive, elaborate or more sophisticated technology. Some people [Sharifabdi 02] have found that coaching is important in extreme programming. It is important here to remember that this coaching is not a continuing cost, given the fact that extreme programming promotes the even development of skills among the members of the team, through practices like pair programming and collective ownership. Beck [Beck 99-2] suggests that the leader of the team should be rotated after certain periods of time. Based on this assumption, it is expected that after a certain period of time any member of the team should be able to coach the team.

In general terms, the validation of the equality side of the null hypothesis should not be seen as an equal situation between the treatments but as a fertile field of opportunities for this young approach.

7. Conclusion

The objective of the experiment was to assess extreme programming. With this purpose, it was compared with a traditional approach which played the role of a control treatment. The observable practices followed by the teams in extreme programming treatment were: planning game, testing, pair programming, simple design, coding standards, collective ownership, continuous integration, small releases, and some cases of metaphors and refactoring. They did not follow "40 hours week" nor "on site customer". The teams followed an additional practice: testing based on requirements.

In this experiment, the null hypothesis was accepted, and the alternative hypothesis was rejected. It means that the results supported the fact that extreme programming teams produced as good results as the traditional approach. The implications of this result are very important. The most relevant one for the Software Engineering community is that a procedure free of Design provides as good results as one including Design. The lack of Design resulted from applying extreme programming.

Internal quality and external quality are unrelated. The behaviour of the internal quality factors was not related to the behaviour of the internal factors. This means that some systems could present

good user characteristics and poor internal construction, or good internal construction and poor presentation for the user, or any other combination. But there was not any pattern, according to the data from the correlation coefficient.

The rejection of the alternative hypothesis and the similar results (final product) obtained from the treatments of the experiment should not be seen as negative result, particularly given the maturity of the processes (traditional approach is more mature than extreme programming), and the cost of the technology required (traditional approach is more expensive than extreme programming).

Acknowledgements

We would like to acknowledge our colleagues Philip McMinn and Haralambos Mouratidis for their collaboration in this project. We should also like to thank our clients who agreed to working with our students on their problems. Macias thankfully acknowledges the support of CONACYT (Mexico) and also the patient help of Anabel Blasco Moreno, Diana Ridley and Sandra Tassie.

Bibliography

[Basili 86] V. R. Basili, R. W. Selby, D. H. Hutchens; Experimentation in software engineering; IEEE Transactions on software engineering, vol. SE-12, pp. 733-743; Jul. 1986.

[Basili 88] V. R. Basili, H. D. Rombach; The TAME Project: Towards Improvement-Oriented Software Environments; IEEE Transactions on software engineering, 14(6):758-773; Jun. 1988.

[Beck 98] K. Beck; Extreme Programming: A Humanistic Discipline of Software Development; Lecture Notes in Computer Science 1382:1-16; 1998.

[Beck 99-1] K. Beck; Embracing change with extreme programming; Computer 32(10):70-77; Oct. 1999.

[Beck 99-2] K. Beck; 1999; Extreme Programming Explained: Embrace Change; Addison-Wesley; U.S.A.; p.t. 190.

[Fowler 01] M. Fowler; Avoiding Repetition; IEEE Software 18(1):97-99; Jan-Feb 2001.

[Holcombe 01-1] M. Holcombe, M. Gheorghe, F. Macias; Teaching XP for real: Some initial observations and plans; Proceedings of 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001); Sardinia, Italy, May 20-23, 2001; 14-17.

[Kitchenham 01] B. A. Kitchenham, S. L. Pfleeger, *et al.*; Preliminar guidelines for empirical research in software engineering; Institute for Information Technology, National Research Council of Canada; Canada, Jan 2001.

[Macias 02] F. Macias, M.Holcombe, M. Gheorghe, "Empirical experiments with XP" in Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 26-30, 2002, 225-228.

[Montgomery 01] D. C. Montgomery; 2001; Design and Analysis of Experiments; 5th Ed.; John Wiley & Sons, Inc.; U.S.A.; p.t. 684.

[Moore 02] I. Moore, S. Palmer, "Making a Mockery" in Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 26-30, 2002, 6-10.

[Putnam 02] D. Putnam, "Where has all the management gone?" in Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 26-30, 2002, 39-42.

[Rumpe 02] B. Rumpe, P. Scholz, "A manager's view on large scale XP projects" in Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 26-30, 2002, 160-163.
[Sharifabdi 02] K. Sharifabdi, C. Grot; Team Development and pair programming -tasks and challenges of the XP coach; in Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 26-30, 2002, 166-169.
[Shepperd 95] M. Shepperd; 1995; Foundations of Software Measurement; Prentice Hall; England; p.t. 234.
[Williams 00] L. Williams, R. K. Kesler, W. Cunningham, R. Jeffreis; Strengthening the case for pair programming; IEEE Software 17(4):19-25; Jul-Aug 2000.
[Wright 02] G. Wright, "eXtreme Programming in a hostile environment" in Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 26-30, 2002, 48-51.

10. Appendix

I	II	III	IV	V	VI	VII	VIII	IX
C	1		46	91	141	132	86	51
B	2		122	1	131	38		60
D	3		78	44	235	83	29	56
D	4	12	52	81	209	125	94	91
A	5		85	36	413	84	43	161
B	6	10	91	48	264	83	179	112
A	7		48	11	115	142	92	63
A	8		94	49	96	135	107	69
C	9		172	25	266	106	26	
D	10		58	41	195	169	4	60
C	11		74	110	74	2	4	66
B	12	55	14	61	123	62	5	129
C	13	6	106	56	459	31	27	32
B	14		44	62	192	64		100
D	15	13	50	11	205	21	7	21
D	16		32	53	195	12		76
B	17		93	121	221	45	9	100
A	18		30	9	8	2	3	13
C	19	44	37	65	44			
A	20		69	77	107	55		150

Time spent in hours

I) Client II) Team label III) Research
IV) Requirements V) Specification/Design VI) Coding
VII) Testing VIII) Review IX) Other

n.b. Teams 1 to 10 applied extreme programming. Teams 11 to 20 applied traditional approach.

I	II	III	IV	V	VI	VII	VIII	IX	X	XI
1	3	3	2	4	4	4	5	5	4	4
2	4	5	4	2	4	2	5	5	2	3
3	3	5	5	5	0	4	5	3		8
4	5	3	2	5	1	2	5	4		7
5	5	5	5	5	4	5	5	4	5	5
6	3	3	3	3	4	3	3	2	2	3
7	3	3	2	3	3	3	4	2	5	3
8	3	3	3	3	3	3	3	3	3	3
9	4	4	4	4	3	4	5	4	5	4
10	3	3	5	5	2	2	5	3		7
11	2	2	2	2	3	3	2	2	4	3
12	3	3	3	3	4	3	4	2	3	4
13	3	2	3	3	3	3	5	2	4	3
14	4	4	3	5	3	5	4	1	4	5
15	3	4	2	5	2	3	5	2		7
16	4	5	3	5	2	3	5	3		7
17	4	4	3	5	4	5	4	2	2	4
18	2	2	2	3	3	3	3	2	3	2
19	4	4	4	4	5	4	5	5	5	4
20	3	4	3	4	4	3	4	4	4	4

External quality factors.

I) Team label
II) Presentation
III) User manual
IV) Installation guide
V) Easy of use
VI) Error handling
VII) Understandable
VIII) Base functionality
IX) Innovation
X) Robustness
XI) Happiness

n.b. Client NHS Cancer Screening Program did not evaluate Robustness. All the marks were ranked from 0 to 5 except in Happiness for NHS Cancer Screening Program teams. These were ranked from 0 to 10.

I	II	III	IV	V	VI	VII	VIII
1	3.5	3	3	6	8	3	4
2	4	4	3	6	7	4	7
3	4	3	3	5	7	4	7
4	4	3	3	7	8	4	8
5	4	4	4	7	6	5	8
6	4	4	4	6	7	4	8
7	4	4	4	7	7	4	8
8	4	4	4	7	4	3	7
9	4	4.5	4.5	8	6	4.5	8
10	3	4	4	8	7	4	8

Internal quality factors. Extreme programming teams.

I) Team label
II) Requirements document
III) Test cases
IV) Test management process
V) Test results (ranked from 0 to 10)
VI) Coding standards (ranked from 0 to 10)
VII) User documentation
VIII) General commentary (log, milestones, strategies, description, plan; ranked from 0 to 10)

I	II	III	IV	V	VI	VII
11	4	7	4	6	4	6
12	4	6	6	8	4	9
13	4	7	8	7	4	7
14	4	6	6	6	4	10
15	4	6	6	7	3	6
16	3	6	7	7	4	7
17	4	6	8	6	4	6
18	4	5	6	5.5	3	7
19	4	2	9	7	4	8
20	4	7	7	8	4	7

Internal quality factors. Traditional approach teams.

I) Team label
II) Requirements document
III) Detailed design (ranked from 0 to 10)
IV) Test results (ranked from 0 to 10)
V) Coding standards (ranked from 0 to 10)
VI) User documentation
VII) General commentary (log, milestones, strategies, description, plan; ranked from 0 to 10)

I	II	III	IV	V	VI	VII	VIII	IX	X
1	12	8	6	2	3	4	1	1	62
2	30	3		2	9	3	3	4	227
3	23		5	2	2	2	1		67
4	9	2		3	4	2	2	1	18
5	45	3	4	3	4	2	4	3	458
6	30	8	2	4	4	1	3	3	265
7	45	4	1	3	2	2	2	1	131
8	44	18	9	3	3	2	2	2	105
9	23	19	3	3	3	5	2	2	254
10	34	8	4	4	6	2	2	2	155
11	29	5	3	3	4	2	2	3	32
12	20	7	7	4	3	3	2	3	119
13	15	10	5	4	4	3	5	4	127
14	22	6	5	3	3	2	2	3	153
15	9	7	7	3	7	2	4	3	91
16	11	7	3	1	2	3	2		45
17	17	3	5	1	5	3			242
18	21	7	6	4	5	3	3		33
19	41	20	17	5	5	2	3	9	235
20	42	8	14	4	2	2		1	73

Size of the projects. Number of requirements and test cases.

I) Team label
II) High priority functional requirements
III) Medium priority functional requirements
IV) Low priority functional requirements
V) Reliability requirements
VI) Usability requirements
VII) Efficiency requirements
VIII) Maintainability requirements
IX) Portability requirements
X) Test cases