Alternating Group Coordinator (AGC): An Approach to Improve eXtreme Programming

Hamid Mcheick[1,*] and Hassan Artail[2,*]

[1]*University of Quebec at Chicoutimi, Department of Computer Science and Mathematic, Postal Code G7H2B1, 555 Boulevard de l'Université, Chicoutimi (Québec), Canada*

[2]*Electrical and Computer Engineering Department American University of Beirut, P.O. Box: 11-0236, Riad El-Solh. Beirut 1107 2020, Lebanon*

Abstract: Agile development methods such as eXtreme Programming (XP) are increasingly adopted by software organizations and engineers to access its effectiveness and the benefits it promises. However, XP has some limitations in certain aspects pertaining to inter-group communication and teamwork. This issue is attributed mostly to the isolation among different pair-programmer groups. In this analysis, we study the impact of applying our solution of the Alternating Group Coordinator (AGC) on the effectiveness of XP. After giving an overview of XP and the issue which we address, we describe the solution we devise and the method used to evaluate this solution through a statistical questionnaire and using it to develop a mathematical model that describes it.

Keywords: Extreme programming, agile methods, inter-group coordination, SE case studies.

INTRODUCTION

The pressure of adapting to customer changing requirements, producing reliable code, and doing this within constrained time schedules are driving software developers and project managers to devise techniques and solutions which are proven to be indispensable for other developers. One such technique that concerns rethinking and reorganizing the Software Engineering process is the eXtreme Programming (XP) methodology. This method has gained wide adoption and is being practiced in academic and industrial settings for various project sizes [1]. XP is tailored to meet the needs of adapting to continuously-changing user requirements and the lack of complete project specifications during the initial stages of the project [2]. Originally, XP was developed by Kent Beck in October 1999 at Chrysler Corporation while working on the C3 project to reduce the software production cost [3]. It's based on 12 key practices that characterize it and distinguish it from other software engineering methods [4]. Theses practices are described in the next page.

Since its inception, XP has seen several updates. Nevertheless, in order to additionally increase its productivity and efficiency, some of the XP practices need further development in certain aspects mostly pertaining to inter-group communication and teamwork. That is, despite its most important characteristic in adapting to the user changing requirements and the high quality code produced from the perspective of the customer, and in emphasising deliverables and milestones [2, 4], XP does not scale properly due to communication overhead and requires coaching before it can be fully adopted [5]. Moreover, it does not emphasize the importance of documentation as in traditional development methods [6], and it does not provide clear mechanisms for programmers to communicate among each other as well as between programmers and customers who in turn become depressed about the lack of clear progress [7]. Also, some drawbacks relating to pair programming have been observed in practice such as the requirement of large blocks of uninterrupted time [8, 9].

This paper attempts to analyze and resolve problematic areas associated with certain aspects of XP affecting productivity and efficiency as part of a complex dimension uncovered in a qualitative approach supported by quantitative modeling. In our analysis we will setup an experiment on two groups of third and fourth year (senior) undergraduate students who had previous experiences in software development. We then develop a mathematical model to analyze and generalize the outcome of this experiment.

The rest of paper is organized as follows. The remainder of this section reviews the rules and practices of XP as well as the problem we are addressing. The next section describes our proposed Alternative Group Coordinator (AGC) solution, and the experiment plus its results. The section that follows presents qualitative and quantitative analysis models that characterize the outcome of this experiment. Next, we present a case study for evaluating the efficiency of the AGC method based on our experience in developing software. Finally, in the last section, we conclude the paper and discuss future work.

MAIN RULES AND PRACTICES OF XP

There are twelve key practices in XP [3, 4, 10], which we discuss here for completeness:

*Address correspondence to these authors at the University of Quebec at Chicoutimi, Department of Computer Science and Mathematic, Postal Code G7H2B1, 555 Boulevard de l'Université, Chicoutimi (Québec), Canada; Electrical and Computer Engineering Department American University of Beirut, P.O. Box: 11-0236, Riad El-Solh. Beirut 1107 2020, Lebanon; Tel: 418-545-5011; E-mails: Hamid_mcheick@uqac.ca; hartail@aub.edu.lb

1874-107X/09

- *The Planning Process*: The XP planning process allows the XP "customer" to define the business value of desired features, and uses cost estimates provided by the programmers, to choose what needs to be done and what needs to be deferred. The effect of XP's planning process is that it is easy to steer the project to success.
- *Small Releases:* XP teams put simple systems into production early, and update them frequently in a very short cycle.
- *Metaphor:* XP teams use a common "system of names" and a common system description that guides development and communication.
- *Simple Design:* A program built with XP should be the simplest program that meets the current requirements. There is not much building "for the future". Instead, the focus is on providing business value. Of course it is necessary to ensure that there is a good design, and in XP this is brought about through "refactoring", discussed below.
- *Testing*: XP teams focus on validating the software at all times. Programmers develop software by writing tests first, and then software that fulfills the requirements reflected in the tests. Customers provide acceptance tests that enable them to be certain that the features they need are provided.
- *Refactoring* [11]: XP teams improve the design of the system throughout the entire development. This is done by keeping the software clean: without duplication, with high communication, simple, yet complete.
- *Pair Programming*: XP programmers write all production code in pairs, two programmers working together at one machine. Pair programming has been shown by many experiments to produce better software at similar or lower cost than programmers working individually.
- *Collective Ownership:* All the code belongs to all the programmers. This lets the team go at full speed, because when something needs to be changed, it can be done without delays.
- *Continuous Integration:* XP teams integrate and build the software system multiple times per day. This keeps all the programmers on the same page, and enables very rapid progress. Perhaps surprisingly, integrating more frequently tends to eliminate integration problems that plague teams who integrate less often.
- *40-hour Week:* Tired programmers make more mistakes. XP teams do not work excessive overtime, keeping them fresh, healthy, and effective.
- *On-site Customer:* An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions as the programmers have them. The effect of being there is that communication improves, with less documentation - often one of the most expensive parts of a software project.
- *Coding Standard:* For a team to work effectively in pairs, and to share ownership of all the code, all the programmers need to write the code in the same way, with rules that make sure the code communicates clearly.

THE PROBLEM

From this quick overview of XP, one can infer the lack of stressing explicit interaction and communication between the members of the group using this method. This problem could become severe at the advanced stages of a project, where project complexity and inter-modular dependencies increase, and at some point during the lifecycle the allocated tasks may no longer be independent among programming pairs.

This issue has motivated our work and drove us to look for a solution that is feasible and could work for all the members of the project team who are using XP. This is the Alternating Group Coordinator (AGC).

ALTERNATING GROUP COORDINATOR (AGC)

This suggestion renders a more *centralized* aspect to the mainly *distributed* XP nature. It adds stability, time saving, and a factor of strength. We recommended that this supervising entity will enjoy administrative expertise for configuring and setting up an efficient production environment. Working with this liaison resembles sending a scout ahead, the AGC, to minimize any roadblocks ahead of time. His/her role will entail the organization of functional compatibility.

Pair programming, frequent partner swapping, and partner mixing, command great merit in XP. As a matter of fact, two programmers working together generate an increased volume of superior code, compared with the same two programmers working separately. Yet, coordination among the different teams forges a problem which routinely disrupts pairing.

With this new scheme, pair programming should produce positive short-term and long-term results. Consequently, one can achieve rapid code generation with redundant expertise and an error filter in the coding phase and enable the cross-pollination of skills that the group needs for the long haul. Our idea introduces the random selection of a team of two programmers already working on the software project to play the role of the AGC for a specified period of time. In this way, the flat hierarchy that basically characterizes XP is maintained with an additional *centralized flavor*. This centralized component sustains flexibility in order to avoid dependencies and possible failure of the project. The elected coordinating team will keep the other pairs more up to date in terms of the current progress of the project. An evident outcome of this scheme would be to motivate the programmers as they participate in the management process and perhaps receive a slight financial bonus in return for the extra work they are providing.

This idea could be emphasized by the actual rules that promote XP: **interactive communication**. The elected team could make use of white boards, positioning and sharing of desk facilities, and stand-up meetings to coordinate the progress of the other teams. On the other hand, the job of the AGC could be facilitated by the use of documentation, which is not advocated by XP. Hence, their stand-up meetings will take much less time and the self sufficiency of the other pair programmers is protected. The customer can also help with the regular production and revision of written requirements, commentary, and the review of work in progress, as well as the ongoing editing of the team's documentation to synchronize results with the customer's efforts.

Each working pair is composed of a coder and a tester. According to Beck, the latter is "*responsible for helping the customer choose and write functional tests*" and running them regularly. The tester seems to be the most appropriate person to take on the job of analyst responsible for documentation and management. In this proposed XP perspective, documentation would enhance the project's scope and efficiency. Continuous design changes would have been hopelessly inefficient without the attributes of this new scheme: the perpetual backtracking, review, and the ability to organize the different programmers contributing to the design effort.

EXPERIMENT SETUP

To test the effectiveness of our proposed change to XP, we used the Software Engineering four-month course given during the spring term of 2008 at the American University of Beirut to third year Electrical and Computer Engineering students. As part of the course requirements, students are supposed to complete a practical software application development project for a real organization. Students had to actually meet with "customer" representatives to get requirements, discuss progress, and hold demonstrations (an average of four meetings per group during the semester). They were divided into groups of 6 and required to apply the software engineering methodology and skills acquired during the course. Students generally had the same educational background as they were majoring in the Computer and Communication Engineering (CCE) program. They were fully aware of how important it is to manage time, space, and other resources in order to achieve a successful job, and most importantly, deliver the project by the assigned deadline. Consequently, there was a grave need to develop techniques within the group to combat possible failures. However, students had no prior knowledge of extreme programming practices or other methodologies (although they were proficient in programming, most notably C++). For this purpose, they started developing techniques that could help them achieve their objectives in a synchronized manner.

Having established the idea of improving extreme programming through the use of the alternating group coordinator (AGC) technique, we asked two of these groups to analyze this idea. Actually, both of these groups used pair programming informally without having prior knowledge about it being one of the building block of extreme programming.

The first group (Group 1) worked on part of a final year project that was carried out by three fourth-year engineering students. The work of Group1 was coordinated by two of the three students working on their final year project (one at a time). Consequently, Group 1 conducted their project while applying the improvement that we introduced to extreme programming; that is, they had two alternating coordinators who had knowledge of what the subgroups are doing and how the overall project is progressing. On the other hand, the second group (Group 2) on which we conducted the experiment had no centralized coordination. Both of the groups were working on equally important projects, and were desperate for ideas that could enhance the progress of their work.

Functional compatibility, rapid code generation, up-to-date feedback, and financial bonus (in terms of time and space resources) were compared between the two groups. Efficiency of the idea was analyzed based on results of statistical and mathematical models.

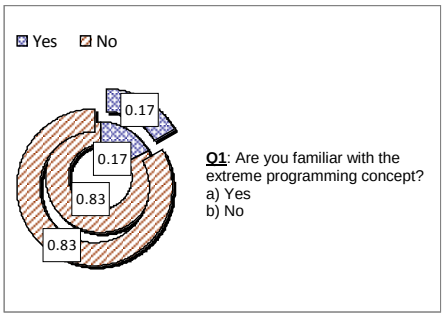
Our methodology for measuring effectiveness relied mostly on asking each member of the two groups a set of question, some of which were common to the two groups, while others were specific to each group, based on the quality of coordination they had received. The questions were divided into two sets. The first set included common questions to both groups and consisted of 12 questions (Questions 1 through 12), while the second set included few group-specific questions: Question 13 for Group 1 and Questions 14 through 17 for Group 2. In turn, the first set comprised two subsets: one about the students background as it relates to the general aspects of XP (Questions 1 through 8); and another subset that is specific to our introduced change, i.e., AGC (Questions 9 through 12).

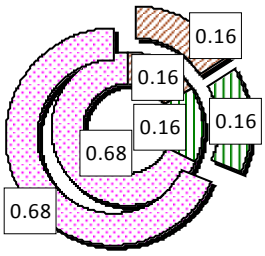
The objective behind asking the first subset of the first set of questions along with the second set was to provide a qualitative analysis of the students' background and knowledge. Before revealing these questions to the two groups, no one had yet been introduced to XP through the course itself. On the other hand, the goal from asking the second subset of the first set was to provide a quantitative analysis. The difference here is that before revealing the questions to the students, the concept of XP and that of the proposed idea for improving XP were explained to them. Lastly, we note that in our quantitative analysis we make use of mathematical models driven from the domain of information theory.

EXPERIMENT RESULTS

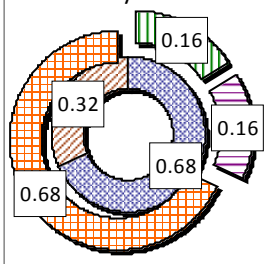
In this section we present the results that represent percentages of answers to each possible answer for the particular question, and then we elaborate on them in the next section. For the first set of questions, each graph includes two donuts to depict the percentages. The inner donut corresponds to Group 1 while the outer donut corresponds to Group 2. For enhanced readability of the results, we include the questions and the corresponding possible answers in the graphs themselves. For the second set of questions, naturally each of the graphs includes a single donut to illustrate the answers. Finally, we note that we did not assign figure numbers to the graphs since we can refer to them by question number.

Common Questions

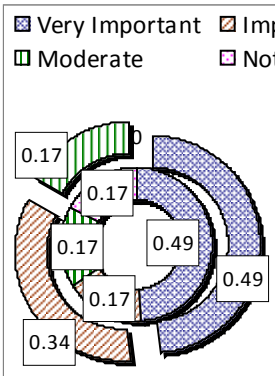




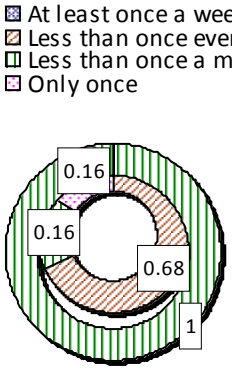
Q2: How much effective is pair programming in a distributed environment?
(a) Highly effective
(b) Effective
(c) Moderate
(d) Not Effective



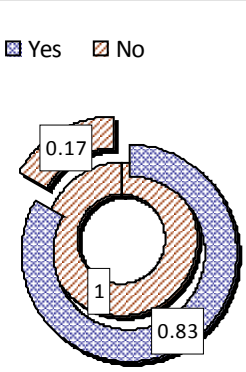
Q6: Rate the overall efficiency of individual parts combination during your software development?
a) Highly effective
b) Effective
c) Moderate
d) Not Effective
e) Very Bad



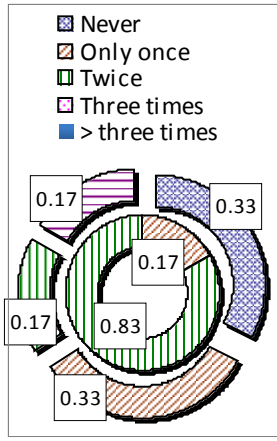
Q3: Rate the importance of group communication in software development, in your opinion.
(a) Very important
(b) Important
(c) Moderate
(d) Not important



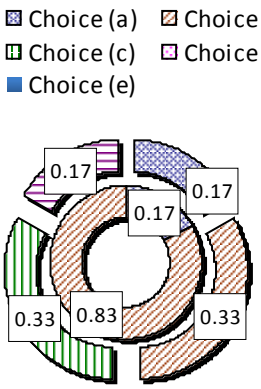
Q7: During the process of development, how often did you have meetings with the entire group members to discuss the overall progress?
(a) At least once per week
(b) Less than once every two weeks
(c) Less than once every month
(d) Once during the sequence of development of the code
(e) Never



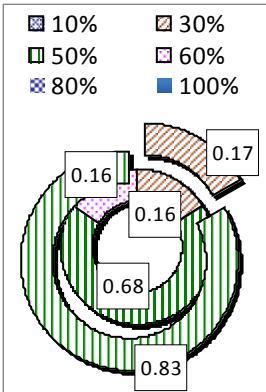
Q4: Did you have problems in coordination within your group?
a) Yes
b) No



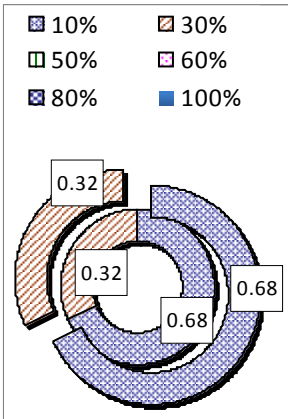
Q8. While working on the project, how often during the sequence of development of the project have you looked at the code of the other groups in order to evaluate the overall project functionality?
a) Never
b) Only once
c) Twice
d) Three times
e) More than three times



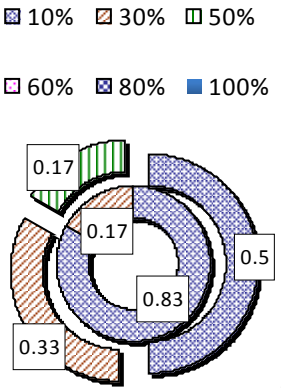
Q5: How can you describe the difficulties you had when trying to combine the individual parts of your project?
a) Time Consuming
b) Causes a decline in the overall quality of the project
c) Parts of the written code turned out to be useless
d) Big parts of the code need to be rewritten.
e) There were no difficulties



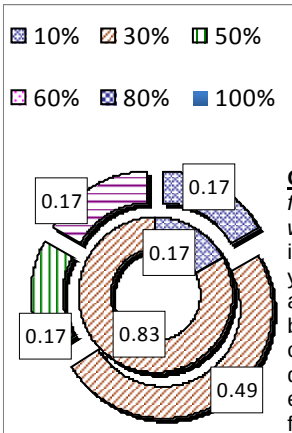
Q9: AGC is thought to add *functional compatibility* to XP programming. How much do you think the opportunities out of 100.0 that this goal is achieved with AGC?
a) 10%
b) 30%
c) 50%
d) 60%
e) 80%
f) 100%



Q10: AGC is thought to provide *rapid code generation* to XP programming. How much do you agree with this (out of a 100)?
a) 10%
b) 30%
c) 50%
d) 60%
e) 80%
f) 100%



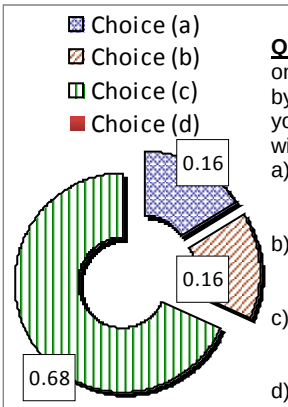
Q11: AGC is thought to provide an *up-to-date feedback* for the project progress in XP programming. How much do you think the opportunities out of 100.0 that this goal is achieved with AGC?
a) 10%
b) 30%
c) 50%
d) 60%
e) 80%
f) 100%



Q12: AGC is thought to provide a *financial bonus (in return of extra work)* for the software developers in XP programming. How much do you agree with this (out of a 100)?
a) 10%
b) 30%
c) 50%
d) 60%
e) 80%
f) 100%

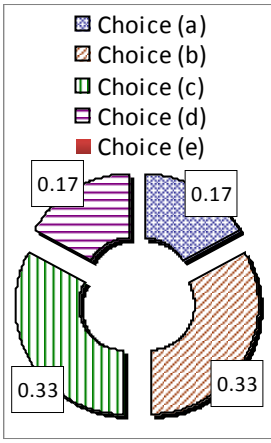
Group Specific Questions

Group 1

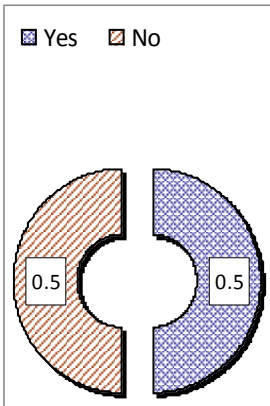


Q13: Suppose you weren't working on a project that was coordinated by a certain person, how would you have managed coordination within your team?
a) We would have asked an outside person to supervise our work
b) We would have assigned one member of the group to coordinate our work
c) We would have assigned two members of the group to coordinate our work
d) We wouldn't have needed a coordinator

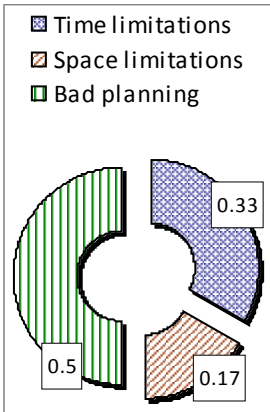
Group 1



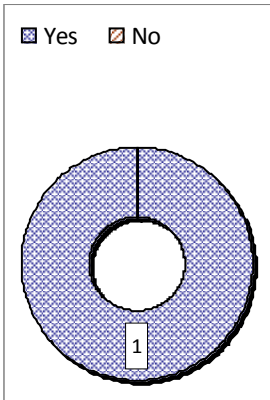
Q14: What problem(s), if any, have you encountered while working in a distributed environment?
a) Difficulty in finding common structure for the final project.
b) Increased complexity at the end, i.e. when trying to combine the individual parts of your project.
c) Inter-groups disagreement and dissatisfaction
d) Increased work pressure with advent of the deadline, i.e. when it is time to combine individual project parts
e) No problems



Q15: Have you thought of assigning a coordinator to ensure interactive communication among all the groups?
a) Yes
b) No



Q16: If yes, what limitations prevented you?
a) Time limitations
b) Space limitations
c) Bad planning



Q17: If you were to redo this project, would you assign a coordinator?
a) Yes
b) No

ANALYSIS OF THE RESULTS

Qualitative Analysis

When analyzing the first set of questions for both groups we can deduce the following:

- Both groups were not initially familiar with extreme programming.
- Both groups were not in favour of pair programming in a distributed environment.
- Both groups had a positive attitude toward the idea of communication within a group.
- Group 2 faced all kind of problems expected in a distributed environment. These problems included difficulty in finding common structure for the final project, increased complexity at the end, i.e., when trying to combine the individual parts of the project, disagreements and dissatisfaction among groups, and increased work pressure with approaching deadlines, i.e. when it is time to combine individual project parts.
- In contrary to Group 1, Group 2 faced problems in coordination within the group.
- Almost all members of Group 1 had no problem when they tried to combine the individual parts of the project. However, the members of Group 2 described this task as time consuming, non-effective, and non-efficient. Group 2 had to rewrite code and spend more time on refactoring during the combination of the code.
- Both Groups did not spend much time in meetings and reviewing each other's code. However, this had more negative effects on Group 2 than Group 1. This is attributed to the fact that Group 1 was coordinated through the AGC who was providing each pair a feedback of the other pairs' work.
- Half of the members of Group 2 were eager for a member to operate as an AGC, but bad planning and time limitations prevented them from doing so.
- Members of Group 1 would certainly assign at least one member to operate as an AGC if there was no coordinator in the project.
- After explaining the concept of AGC to Group 2, we asked the members of the group if they were to redo their project and assign a coordinator. The answer was definitely yes.

The outcome was that Group 1 excelled over Group 2 in terms of performance and satisfaction due to the presence of the AGC.

Quantitative Analysis

Only one member of the two groups had an idea about eXtreme Programming. Therefore, we had to describe this method as well as our proposed improvement to the groups. This enabled them to apply XP and afterwards answer Questions 9 through 12 of the questionnaire.

Mathematical Analysis of the Gathered Data

By making use of some statistical tools that are based on the domain of information theory, we evaluate the application of the survey to the proposed AGC. The theory of entropy, relative entropy, and mutual information are defined as functions of probability distributions. They characterize the behaviour of random variables and allow us to estimate the probabilities of rare events (large deviation theory) and find the best error exponent in hypothesis tests. Hence, it is both appropriate and perhaps necessary to briefly describe these probabilistic concepts.

Entropy

The entropy, as a concept, is a measure of the uncertainty of a random variable. Let X be a discrete random variable with alphabet χ and probability mass function:

$$p(x) = Pr(X=x),\ x \in \chi.$$

Then the entropy $H(X)$ of the discrete random variable X is defined by:

$$H(X) = -\sum p(x) \log [p(x)].$$

The log here is base 2, and entropy is expressed in bits. As we have noted, entropy is a function of the distribution of X. It does not depend on the actual values taken by the random variable, but only on the probabilities.

Next, we examine two useful properties of the entropy function, and then provide a proof to the second one since this will give an insight into this important measure:

- Since $0 \leq p(x) \leq 1$, so $log\ (1/p(x)) \geq 0$. It follows that $H(X) \geq 0$.
- $H(X) \leq log|\chi|$

Proof

To begin the proof, it is necessary to introduce the concept of relative entropy between two probability distributions p and q. This is given by:

$$D\ (p//q) = \sum p(x)\ log\ [p(x)/q(x)]$$

Which can be shown to be positive 0 based on Jensen's inequality: for a concave function f and any random variable X, $E[f(X)] \leq f(E[X])$, where the notation $E[X]$ means the expected value of X. Then, if we assume q to be a uniform distribution over χ then:

$$q(x_i)=1/|\chi| \text{ where } i=1,2,\ldots,|\chi|.$$

Thus we get:

$$\begin{aligned} D(p//q) &= \sum p(x) log [p(x)/q(x)] \\ &= \sum p(x)\ log\ [p(x)] - \sum p(x)\ log\ [q(x)] \\ &= -H(X) - \sum p(x)\ log\ [1/|\chi|] \\ &= -H(X) - log\ [1/|\chi|] \sum p(x) \\ &= -H(X) - log\ [1/|\chi|], \end{aligned}$$

and since $D\ (p//q) \geq 0$, *then*,

$H(X) - log\ [1/|\chi|] \geq 0$, and consequently,

$H(X) \leq - log\ [1/|\chi|]$, or $H(X) \leq log\ (|\chi|)$

We note also that $D\ (p||q) = 0$ in the case the original assumption hold, that is if q is uniform, which occurs if and only if $p=q$. Thus at that point $H(X)=log(|\chi|)$ and p is a uniform distribution.

Applications of the Entropy Function to the Effect of AGC

After introducing the concept of the entropy function and the main properties that are useful in evaluating AGC, we try to derive the probability distribution of the effect of the AGC on the following variables:

- Adding functional compatibility
- Rapid code generation and error filtering in the coding phase
- Members are up to date on the progress of the project
- Financial bonus for the extra work the AGC is doing.

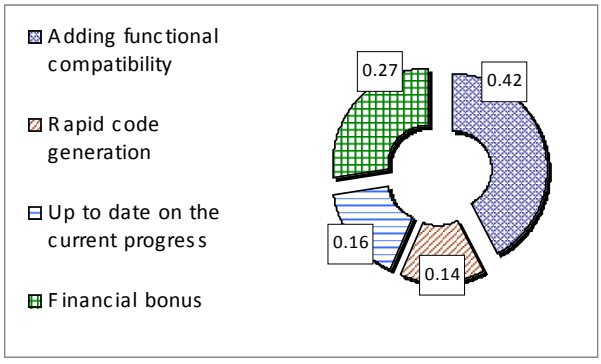
The following chart reflects the responses of both groups of students. The four values were obtained from the answers of Questions 9, 10, 11, and 12 (as implied by the name of each shown category in the graph below). Basically, each shown value was computed as follows (refer to the charts of Questions 9 through 12):

Considering a particular category c (e.g., "Rapid Code Generation" corresponding to Question 10), if p_{ci} represents percentage i, which is one of the answers (i.e., 10%, 30%, 50%, 60%, 80%, or 100%), and a_{gci} is the corresponding answer (as shown in the figures of Questions 9 through 12), where the subscript g relates to the group number, then the value v_c for this category (as shown below in the graph) can be computed as follows:

$$v_c = \frac{\sum_{g=1}^{2} \sum_{i} p_{ci} a_{gci}}{2}$$, where again c refers to Question 9, 10, 11, or 12.

Next, we normalize each value to get a sum that is equal to 1. Specifically:

$v_c = v_c / (v_9 + v_{10} + v_{11} + v_{12})$, where the numeric subscripts were used to relate each value to one of the four questions (i.e., v_c can be v_9, v_{10}, v_{11}, or v_{12}).



In our application, the alphabet χ will be:

χ = { - adding functional compatibility,

- rapid code generation and error filtering in the coding phase,
- up-to-date member awareness of the current progress of the project,
- financial bonus for the extra work as the AGC}

For simplicity, we assume that $\chi=\{X_1, X_2, X_3, X_4\}$, where X_1, X_2, X_3 and X_4 are the four elements listed above.

From the chart above, we infer the following:

- The probability of the occurrence of X_1 due to implementing the AGC approach is 0.5
- The probability of the occurrence of X_2 due to AGC is 0.1
- The probability of X_3 being true is 0.1
- The probability of X_4 happening is 0.3

Hence:

$P_X(x)$ = { $\{0.5 \text{ if } x=X_1\}$
$\{0.1 \text{ if } x=X_2\}$
$\{0.1 \text{ if } x=X_3\}$
$\{0.3 \text{ if } x=X_4\}$ }

Then we can write:

$H(X) = -\sum p(x) \log [p(x)]$

= -0.5log0.5 -0.1log0.1 -0.1log0.1 -0.3log0.3

= 1.685 bit.

But since $|\chi| = 4$, and so $\log |\chi| = 2$ (remember, this is log-base 2). Therefore, $H(X)$ is close to $\log |\chi|$, which shows that χ has a close distribution to the uniform one. This in turn illustrates that once we ensure the presence of the AGC in some entity practicing the XP, a great advantage will be reached in a quasi-uniform fashion with respect all the variables in question. As a result, this will offer a better approach to doing the work, and should improve quality, job satisfaction, productivity, and predictability.

CASE STUDY

Following the results of the experiment above, which were in favour of the AGC idea, the next step was to conduct a case study on Group 1, mentioned above, to evaluate the efficiency of AGC in software development based on a limited test case.

The customer in this case was a local distributor of dairy products. The software was to create a web application and a graphical user interface that would make use of certain traveling salesman algorithms to manage a company's limited resources. Thus, the project was made of two parts. The first was about developing the web application and the graphical user interface, while the second part was to encode the available traveling salesman algorithms (based on neural networks techniques) and test them in order to choose the optimum algorithm that would manage the limited resources efficiently. We had two choices: one was to have all the members of the group working on both parts of the project, and the other was to divide students into two groups each working on a part, and then have two of the three senior students (see the earlier description about the composition of the groups) alternate the role of the AGC. After brainstorming both ideas and analyzing the impact on the project schedule and required effort, all members of the group were in favour of the second option.

The authors of this paper have taught Software Engineering multiple times and therefore, they are in a position to comparatively assess the value of using AGC by contrasting the results of this experiment with that of Group 2 and as a matter of fact to those of the other groups:

- Having an AGC coordinating the group noticeable increased the throughput of the developers, as they had to inquire less about every little detail from others working in different subgroups.
- Less time was lost in group meetings because the AGC was providing each subgroup with feedback frequently and on-demand.
- The problems that we encountered in combining the work of the two subgroups were minor, because each subgroup had a "good" understanding of the requirements of the other subgroup.
- A synergy was formed between the two subgroups. This was attributed to the fact that the expectation of each member of the group was outlined and adjusted through the AGC.
- The quality of output was seemingly better. The AGC had the full picture and was the guide who led the project to a successful ending. The approach resulted in worrying less about the work and progress of others.
- With AGC, the developers gained functional compatibility in their work.
- Rapid code generation and errors filtering were uncommon in this particular project.
- All members were up to date on the current progress of the project.

CONCLUSIONS

The eXtreme Programming (XP) methodology has received significant attention from both researchers and practitioners in the business of software development. XP aims to adapt to very dynamic situations, mostly dealing with frequent and unexpected changes in the customer requirement specifications. Nevertheless, XP has certain aspects that could be improved, and one of them deals with inter-group communication and coordination. Our work aimed to address this issue by introducing the concept of Alternating Group Coordinator (AGC) who has the responsibility of keeping the subgroups "on the same page". This scheme introduces some *centralization* to the mainly *distributed* XP nature. We have proven in our limited study that AGC adds stability, time savings, and robustness to the developed code.

An experiment was conducted to analyze the impact of AGC on the software developers. To this end, qualitative and quantitative analyses were performed. The latter involved the development of a mathematical model to study the "measured" data. The model's output was consistent with the conclusions drawn from the qualitative analysis, in that it showed that AGC can improve software quality and productivity.

For future work, we will pursue setting up experiments in concrete industrial settings involving larger groups and different types of projects. Such an expanded study will yield statistically significant results and will serve to increase the confidence in the drawn conclusions about AGC in relation to XP.

ACKNOWLEDGMENTS

This work was sponsored by the Natural Sciences and Engineering Research Council (NSERC) of Canada, by the University Research Board (URB) of the American University of Beirut (Lebanon), and by the University of Quebec at Chicoutimi (Canada).

REFERENCES

[1] O. Olagbegi and H. M. Haddad, "Agile Development: Do advantages outweigh shorts comings?", Proceedings of Sofwtare Engineering Reaserch and Practice (SERP2008-WorldComp08), 2008, pp. 46-52.
[2] J. Newkirk, "Introduction to Agile Processes and Extreme Programming", 24th International Conference on Software Engineering, 2002, pp. 695-696.
[3] K. Beck, *Extreme Programming Explained: Embrace Change*. Addition-Wesley Longman, 2000.
[4] K. Beck, "Embracing Change with extreme programming", *IEEE Computer*, vol. 32, pp. 70-77, October 1999.
[5] M. Muller and W. F. Tichy, "Case Study: Extreme Programming in a University Environment", 23rd International Conference on Software Engineering, May 2001, pp. 537-544.
[6] L. Cao and R. Balasurbramiam, "Agile Software Development: Ad hoc Practices or Sound Principles", *IT Professional*, vol. 9, pp. 41-47, March 2007.
[7] A. Cockburn, *Agile Software Development*. Addison-Wesley, 2007.
[8] S. R. Schach, *Object-Oriented Software Engineering*. McGraw-Hill Companies, 2008.
[9] J. Drobka, D. Noftz, and R. Raghu, "Piloting XP on Four Mission-Critical Projects", *IEEE Software*, vol. 21, pp. 70-75, November 2004.
[10] R. Jeffries, C. Hendrickson, A. Anderson, *Extreme Programming Installed*. Addison-Wesley, 2001.
[11] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1997.

Received: November 24, 2008 Revised: January 01, 2009 Accepted: January 23, 2009