

Social Perspective of Software Development Methods: The Case of the Prisoner Dilemma and Extreme Programming

Orit Hazzan¹ and Yael Dubinsky²

¹ Department of Education in Technology & Science
Technion – Israel Institute of Technology
oritha@tx.technion.ac.il

² Department of Computer Science
Technion – Israel Institute of Technology
yael@cs.technion.ac.il

Abstract. One of the main dilemmas with which software development teams face is how to choose a software development method that suits the team as well as the organization. This article suggests a theory that may help in this process. Specifically, Extreme Programming (XP) is analyzed within the well known framework of the prisoner dilemma. We suggest that such an analysis may explain in what situations XP may fit for implementation and, when it is used, the way it may support software development processes.

Keywords: Agile software development methods, extreme programming, social theories, game-theory, the prisoner dilemma.

1 Introduction

Software development is a complicated task that is strongly based on the people carried it out. The task of deciding upon the appropriate software development method that suits the organization and the team should address the fitness of the method to the people involved (Dines, 2003). Usually methods are selected according to organization and team traditions and according to practical-professional-technical considerations like the programming languages and tools. This situation motivated us to check the analysis of software development methods from other perspectives, such as social and cognitive ones.

This article focuses on a social framework. Specifically, from the social perspective, we apply a game theory framework – the prisoner’s dilemma – which is usually used for the analysis of cooperation and competition. Our arguments are illustrated by Extreme Programming (Beck, 2000). We view the perspective presented in this paper as part of our research about human aspects of software engineering in general, and our comprehensive research about cognitive and organizational aspects of XP, both in the industry and the academia, in particular

(Hazzan and Dubinsky, 2003A, 2003B; Tomayko and Hazzan, 2004; Dubinsky and Hazzan, 2004).

2 A game theory perspective: The prisoner's dilemma¹

Game theory analyzes human behavior by using different theoretical fields such as mathematics, economics and other social and behavioral sciences. Game theory is concerned with the ways in which individuals make decisions, where such decisions are mutually interdependent. The word "game" indicates the fact that game theory examines situations in which participants wish to maximize their profit by choosing particular courses of action, and in which each player's final profit depends on the courses of action chosen by the other players as well.

In this section, a well-known game theory framework is used, namely the prisoner's dilemma. This framework illustrates how the lack of trust leads people to compete with one another, even in situations in which they might gain more from cooperation. In the simplest form of the prisoner's dilemma game, each of two players can choose, at every turn, between cooperation and competition. The working assumption is that none of the players knows how the other player will behave and that the players are unable to communicate. Based on the choices of the two players, each player gains points according to the payoff matrix presented in Table 1, which describes the game from Player A's perspective. A similar table, describing the prisoners' dilemma from Player B's perspective, can easily be constructed by replacing the locations of the values 10 and (-10) in Table 1.

Table 1. The prisoner's dilemma from player A's perspective

	B cooperates	B competes
A cooperates	+ 5	-10
A competes	+10	-5

The values presented in Table 1 are illustrative only. They do, however, indicate the relative benefits gained from each choice. Specifically, it can be seen from Table 1 that when a player does not know how the other player will behave, it is advisable for him or her to compete, regardless of the opponent's behavior. In other words, if Player A does not know how Player B will behave, then in either case (whether Player B competes or cooperates), Player A will do better to compete. According to this analysis, both players will choose to compete. However, as can be seen, if both players choose the same behavior, they will benefit more if they both cooperate rather than if they both compete.²

As it turns out, the prisoner's dilemma is manifested also in real life situations, in which people tend to compete instead of to cooperate, although they can benefit more from cooperation. The fact that people tend *not* to cooperate is explained by

¹ This analysis is partially based on Tomayko and Hazzan, 2004.

² For more details about the Prisoner's Dilemma, see the GameTheory.net website at: <http://www.gametheory.net/Dictionary/PrisonersDilemma.html>.

their concern that their cooperation will not be reciprocated, in which case they will lose even more. The dilemma itself stems from the fact that the partner's behavior (cooperation or competition) is an unknown factor. Since it is unknown, the individual does not trust her partner, nor does the partner trust her, and, as described in Table 1, both parties choose to compete.

In what follows, the prisoner's dilemma is used to show that competition among team members is the source of some of the problems that characterize software development processes. To this end, it should be noted first that, in software development environments, cooperation (and competition) can be expressed in different ways, such as, information sharing (or hiding), using (or ignoring) coding standards, clear and simple (or complex and tricky) code writing, etc. It is reasonable to assume that such expressions of cooperation increase the project's chances of success, while such expressions of competition may add problems to the process of software development.

Since cooperation is so vital in software engineering processes, it seems that the quandary raised by the prisoner's dilemma is even stronger in software development environments. To illustrate this, let us examine the following scenario according to which a software team is promised that if it completes a project on time, a bonus will be distributed among the team members according to the individual contribution of each team member to the project. In order to simplify the story, we will assume that the team comprises of only two developers – A and B. Table 2 presents the payoff table for this case

Table 2: The prisoner's dilemma in software teams

	B cooperates	B competes
A cooperates	The project is completed on time. A and B get the bonus. Their personal contribution is evaluated as equal and they share the bonus equally: 50% each.	A's cooperation leads to the project's completion on time and the team gets the bonus. However, since A dedicated part of her time to understanding the complex code written by B, while B continued working on her development tasks, A's contribution to the project is evaluated as less than B's. As a result, B gets 70% of the bonus and A gets only 30%.
A competes	The analysis is similar to that presented in the cell 'A cooperates / B competes'. In this case, however, the allocation is reversed: A gets 70% of the bonus and B gets 30%.	Since both A and B exhibit competitive behavior, they do not complete the project on time, the project fails and they receive no bonus: 0% each.

The significant difference between Table 2 and the original prisoner's dilemma table (Table 1) lies in the cell in which the two players compete. In the original table, this cell reflects an outcome that is better for both players than the situation in which

one cooperates and the opponent competes. In software development situations (Table 2), the competition-competition situation is worst for both developers. This fact is explained by the vital need for cooperation in software development processes. It can be seen from Table 2, that in software development environments, partial cooperation (reflected in Table 2 by the cooperation of only one team member) is preferable to no cooperation at all.

Naturally, in many software development environments, team members are asked to cooperate. At the same time, however, they are unable to ensure that their cooperation will be reciprocated. In such cases, even if there is a desire to cooperate, there is no certainty that the cooperation will be reciprocated, and, as indicated by the prisoner's dilemma table (Table 1), each team member will prefer to compete. However, as indicated by Table 2, in software development situations, such behavior (expressed by the competition-competition cell) results in the worst result for *all* team members.

3 Application of the Prisoner Dilemma for the analysis of XP

In a recent interview, Kent Beck³ was asked when XP is not appropriate. Beck answered that "[i]t's more the social or business environment. If your organization punishes honest communications and you start to communicate honestly, you'll be destroyed." As can be seen, Kent talks in terms of punishment. In fact, Kent describes a situation in which a team member cooperates ("communicates honestly") while the environment competes ("punishes honest communications"). In terms of the prisoner's dilemma, this is the worst situation for the cooperator, and so, it is expected that no one will cooperate in such an environment. If the entire organization, however, communicates honestly, that is to say, cooperates, there is no need to worry about cooperating, since it is clear that the other members of the organization will cooperate as well. Consequently, the entire organization reaps the benefits of such cooperation.

In what follows we demonstrate the analysis of XP from the game theory perspective. This analysis illustrates how XP enhances trust among team members and methodologically leads them into cooperation-cooperation situations. Specifically, we explain how, from a game theory perspective, two of the XP values (simplicity and courage) as well as several of the XP practices, lead to the establishment of development environments, the atmosphere of which can be characterized by the cooperation-cooperation cell of the prisoner's dilemma (cf. Table 2).

In the case of the value of *courage*, cooperation implies, among other things, that all team members have the courage to admit and to state explicitly when something goes wrong. From the individual's point of view, this means that no one is conceived of as a complainer if one issues a warning when something goes wrong. As Beck says, from the individual's point of view, a warning is worthwhile only if

³ An interview with Kent Beck, June 17, 2003, *Working smarter, not harder*, IBM website: <http://www-106.ibm.com/developerworks/library/j-beck/>

one knows that the organization encourages such behavior and that the other members of the organization behave similarly. Otherwise, it is not worth expressing courage, as such behavior might be conceived of as a disruption, and eventually one might suffer certain consequences as a result of such behavior. In our case, the organization is the XP development environment. Consequently, all team members are committed to the value of courage, all of them can be sure that they are not the only ones to express courage, and no one faces the dilemma whether to cooperate (that is, to issue a warning when something goes wrong) or not. As described in Table 2, all team members benefit from this cooperation.

With respect to the value of simplicity, cooperation is expressed when all activities related to the software development are conducted in the simplest possible way. Thus, for example, all team members are committed not to complicate the code they write, but rather to develop clear code, which is understood by all team members. As all team members are committed to working according to this norm, the development environment is stabilized within the cooperation-cooperation cell, a less complicated development environment is established (for example, the code is simpler), and all team members benefit.

In what follows, several of the XP practices are analyzed by the prisoner's dilemma framework.

Test-First Programming: Cooperation in this case means writing tests prior to the writing of the code; competition means code writing that is not in accordance with this standard. The rationale behind this practice relates to scope creep, coupling and cohesion, trust and rhythm (Beck, 2005, p. 50-51). Specifically, Beck says: "Trust – It's hard to trust the author of code that doesn't work. By writing clean code that works and demonstrating your intentions with automated tests, you give your teammates a reason to trust you" (Beck, 2005, p. 51). This perspective even fosters the trust element in XP team. Specifically, from the prisoner dilemma perspective, because all team members are committed to working according to XP in general, they are all committed to adopting the practice of test-first programming in particular. Naturally, this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them apply the test-first programming practice. Consequently, they all benefit from the quality of software that is developed according to test-first programming.

Collective Ownership: Since XP is the development environment, all team members are committed to apply all the XP practices and, in particular, the practice of collective ownership. The meaning of cooperation in the case of collective ownership is code sharing by all team members; whereas competition means concealing of information. This, however, is not the full picture. In addition, each team member knows that the other team members are also committed to working according to XP. Specifically, with respect to the practice of collective ownership this means that they are all committed to sharing their code. In other words, the practice of collective ownership implies that since all team members are committed to working according to XP, they all cooperate, and share their codes. Thus, the

unknown behavior of the others, which is the source of the prisoner's dilemma, ceases to exist. Consequently, team members face no (prisoner's) dilemma whether to cooperate or not, and since they are guided by the practice of collective ownership, they all cooperate and share their code with no concern about whether their cooperation will be reciprocated or not. Since, for purposes of software quality, it is required that knowledge be passed on among team members, all team members benefit more from this practice than if they had chosen to be in competition with one another. Thus, the practice of collective ownership yields a better outcome for all team members.

Coding Standards: Cooperation in this case means writing according to the standards decided on by the team; competition means code writing that is not in accordance with these standards. Because all team members are committed to working according to XP in general, they are all committed to adopting the practice of coding standards in particular. Naturally, this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them write according to the code standards. Consequently, they all benefit from the quality of software that is developed according to decided coding standards.

Sustainable Pace: The rationale for this practice is that tired programmers cannot produce code of high quality. Cooperation in this case means that all team members work only 8-9 hours a day; competition is expressed by working longer hours, in order, for example, to impress someone. The fact that all team members are committed to working according to XP, ensures that they all work at a sustainable pace in the above sense, and that no one "cheats" and stays longer so as to cause his or her contribution to be perceived as greater. Consequently, none of the team members is concerned about competing in this sense. Eventually, all team members benefit from the practice of sustainable pace.

Simple/Incremental Design: Cooperation in this case means to "strive to make the design of the system an excellent fit for the needs of the system that day (Beck, 2005, p. 51); competition means for example to add (sometimes) redundant features to the code, thus complicating the design. Because all team members are committed to working according to XP in general, they are all committed to adopting this practice in particular. Naturally, from the prisoner dilemma perspective this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them keep the design simple and incremental. Consequently, they all benefit from the quality of software that is developed according to simple/incremental design.

Pair-Programming: Cooperation in this case means to accept the rules of pair programming, such as switching between driving and navigating; competition means not to accept these rules, for example, to code without a pair. From the prisoner dilemma perspective, because all team members are committed to working according to XP in general, they are all committed to adopting the practice of pair

programming in particular. Naturally, this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them apply the rule of the pair programming practice enabling a constant code inspection. Consequently, they all benefit from the quality of software that is developed according to the practice of pair programming.

We believe that at this stage the readership can apply a similar analysis with respect to the other XP practices. Clues for the need of such analysis of software development environments are presented in previous writings. For example, Yourdon (1997) says that "In the best of all cases, everyone will provide an honest assessment of their commitment and their constraints". (p. 65). Further, Yourdon tells about "Brian Pioreck [who] reminded me in a recent e-mail message that it's also crucial for the team members to be aware of each other's level of commitment, which the project manager can also accomplish through appropriate communication: I think you also have to make their commitments public through the use of project plan. Everyone sees the total involvement of all team members this way and what their own involvement means to the project." The contribution of the **Planning Game** to the achievement of this commitment of all team members is clearly derived from the analysis of the planning game from the prisoner dilemma perspective.

4 Conclusion

Cockburn (2002) says that "software development is a group game, which is goal seeking, finite, and cooperative. [...] Software development is [...] a cooperative game of invention and communication. There is nothing in the game but people's ideas and the communication of those ideas to their colleagues and to the computer". (p. 28). This article also applies a game oriented perspective at software development processes. Specifically, this article uses the prisoner dilemma, a well known game-theory framework, for the analysis of software development methods in general and for the analysis of XP in particular. It is suggested that this article demonstrates how a software development method can be analyzed, not only by referring to its technical benefits but, rather, by suggesting ways in which the software development method is viewed from a social perspective.

The scope of this paper is limited to the examination of a software development method in general and of XP in particular by one framework. It is suggested that similar examinations in at least three directions can be carried out.

First, other software development methods can be analyzed using the prisoner dilemma framework;

Second, the application of other game theory methods for the analysis of software development methods can be checked. Kent, for example, has demonstrated the benefits of Win-Win situations: "Mutual benefit in XP is searching for practices that benefit me now, me later and my customer as well". (Kent, 2005, p. 26). Further, we may analyze situation that should be avoided in software development methods

from a game theory perspective. For example, we may analyze the influence of zero-sum situations of software development processes.

Third, it is worth examining the analysis of software development methods based on additional research frameworks and theories borrowed from other disciplines. For example, currently we explore the application of the cognitive theory constructivism (cf. Piaget, 1977; Davis, Maher and Noddings, 1990; Smith, diSessa and Roschelle, 1993), which examines the nature of learning processes, for the analysis of software develop methods.

References

1. Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.
2. Beck, K. (with Andres, C., 2005, second edition). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.
3. Cockburn, A. (2002). *Agile Software Development*. Addison-Wesley.
4. Davis, R. B., Maher, C. A. and Noddings, N. (1990). Chapter 12: Suggestions for the improvement of mathematics education. In Davis, R. B., Maher, C. A. and Noddings, N. (eds.). *Journal for Research in Mathematics Education, Monograph Number 4, Constructivist Views on the Teaching and Learning of Mathematics*, The National Council of Teachers of Mathematics, Inc., pp. 187-191.
5. Dubinsky, Y. and Hazzan, O. (2004). Roles in agile software development teams, *Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering*, Garmisch-Partenkirchen, Germany, pp. 157-165.
6. Dines, B. (2003). What is a method?, an essay on some aspects of domain engineering. *Monographs In Computer Science - Programming methodology*, pp. 175 - 203.
7. Hazzan, O. and Dubinsky, Y. (2003A). Teaching a Software development methodology: The Case of Extreme Programming, *The proceedings of the 16th International Conference on Software Engineering Education and Training*, Madrid, Spain, pp. 176-184.
8. Hazzan, O. and Dubinsky, Y. (2003B). Bridging cognitive and social chasms in software development using Extreme Programming, *Proceedings of the Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering*, Genova, Italy, pp. 47-53.
9. Piaget, J. (1977). Problems of Equilibration. In Appel, M. H and Goldberg, L. S. (1977). *Topics in Cognitive Development, Volume 1: Equilibration: Theory, Research and Application*, Plenum Press, NY, pp. 3-13.
10. Smith, J. P., diSessa, A. A. and Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences*, **3**(2), pp. 115-163.
11. Tomayko, J. and Hazzan, O. (2004). *Human Aspects of Software Engineering*, Charles River Media.
12. Yourdon, E. (1999). *Death March*. Prentice Hall.