

Bridging Cognitive and Social Chasms in Software Development using Extreme Programming

Orit Hazzan¹ and Yael Dubinsky²

¹ Department of Education in Technology & Science, Technion, Israel

² Department of Computer Science, Technion, Israel

Abstract. Extreme programming (XP) is one of the agile software development methodologies. It achieves its goals by the implementation of twelve practices, all aimed at reducing risks in software development and improving software quality. This paper presents two chasms inherent in software development processes - a cognitive chasm and a social chasm - and describes, based on our experience, how the twelve XP practices can help bridge these chasms.

1 Introduction

Software development is a very sophisticated activity from both a cognitive and a social perspective. Indeed, when examining common problems faced by software developers, both cognitive challenges (such as program comprehension) and social difficulties (such as the need to meet customer requirements or communication problem between teammates) can be found.

By reviewing each of the twelve eXtreme Programming (XP) practices, this paper examines XP from both a cognitive and a social perspective. Specifically, we explain how each of the twelve XP practices helps reduce cognitive and/or social complexities involved in software development.

2 The Abstraction Chasm and the Satisfaction Chasm

In this section, we present two terms that will be used throughout the article: the *abstraction chasm* and the *satisfaction chasm*. The ways by which XP helps bridge these two chasms are examples of how XP practices help reduce cognitive and social complexities of software development. In future work we intend to examine other means by which XP may reduce cognitive and social complexities.

The Abstraction Chasm: In the process of software development, software developers are required to think on various levels of abstraction and move between abstraction levels. In other words, programmer must move from a global view of the system (high level of abstraction) to a local, detailed view of the system (low level of abstraction), and vice versa. For example, when trying to

understand customers' requirements during the first stage of development, developers must have a global view of the application (high level of abstraction). On the other hand, when coding a specific class, a local perspective (on a lower abstraction level) should be adopted. Obviously, there are many intermediate abstraction levels in between these two edges that programmers should consider during the process of software development. However, the knowledge of how and when to move between different levels of abstraction does not always come naturally, and requires some degree of awareness. For example, a programmer may remain in too low a level of abstraction for too long a time, while the problem he or she faces could be solved immediately should the problem be viewed on a higher level of abstraction. The shift to that higher abstraction level might not be made naturally, unless one is aware that this may be a step towards a solution. The chasm in this case is observed between the inherent need in software development to move between local and global views of the application and the tendency to remain on one level of abstraction for too long a time.

In Section 3, we illustrate how XP helps reduce part of the cognitive complexity of software development by guiding programmers in the transition between levels of abstraction. This is referred to, in what follows, as bridging the *abstraction chasm*, the extremes of which are *local* and *global* views of the application.

The Satisfaction Chasm: In the process of software development, software developers must alternate between satisfying their current needs and supporting their teammates. For example, when trying to understand a specific piece of code, the programmer must first concentrate on his or her needs and consider what he or she does not understand; After completing the development of a specific class, developers must then consider their teammates' perspective and check whether the code can be improved in order to make it more accessible to the minds of others who did not participate in its development. However, how and when to alternate between satisfying ones individual needs and those of the others, with respect to software development, is no trivial matter. For example, one may continue coding as long as the code passes all tests, without realizing that the code is incomprehensible to others who did not participate in the development of that piece of code. The chasm in this case is observed between the inherent need to move between considering the individual needs and those of the collective and the tendency to deal only with the individual's needs with respect to software development.

In Section 3 we illustrate how XP helps reduce part of the social complexity of software development by guiding programmers in the transition between satisfying their own needs and considering the needs, perspectives and mental processes of others who are involved in the process of software development. As will be illustrated, some of the XP practices encourage the programmer to postpone satisfaction of immediate personal needs and first to dedicate additional efforts to the satisfaction of collective needs. By doing so, the software becomes more mentally accessible to the entire team, and time, money and work hours are eventually saved. This will be referred to, in what follows, as bridging of

the *satisfaction chasm*, the extremes of which are *individual* and *collective* needs.

3 How Does XP Bridge the Chasms?

This section shows how each of the XP practices helps bridge the abstraction chasm and the satisfaction chasm. Specifically, we will illustrate how XP guides the transition between different levels of abstraction and between the consideration given both to individual and collective needs during the process of software development. Our claim is that while these two transitions should be inherent in any software development process, in many cases the natural tendency of human being is to behave in an opposite manner, i.e. to fail to think on different levels of abstraction and to fail to consider both individual and collective needs. We believe that these two human tendencies are the source of some of the major problems that are common in the world of software development. Furthermore, we believe that the strength of XP is in the fact that its twelve practices provide very specific guidance on how to bridge the abstraction and satisfaction chasms. We suggest that such analysis helps in understanding the strength of XP.

The twelve practices are presented below according to the XP practices mapping suggested by [2]. The mapping highlights both the social and cognitive aspects of XP, and, for each specific XP practice, reflects the level of awareness required for its implementation (see Table 1). Specifically, the twelve XP practices are mapped along two dimensions: 'Aspect' and 'Awareness'. On the 'Aspect' dimension, the XP practices are classified as addressing either a technical aspect or a human aspect of software development. The second dimension, 'Awareness', maps XP practices according to the level of awareness required to implement each of the practices throughout the software development process.

Table 1. Mapping the XP practices according to technical and human aspects & levels of awareness

Awareness	Technical Perspective	Human Perspective
High	Refactoring Simple design	Metaphor Collective ownership
Intermediate	Coding standards Testing	Pair programming Planning game
Low	Small releases Continuous integration	40-hour week On-site customer

In the following explanation, we start with the more easily implemented practices (requiring a low level of awareness, e.g., 40-hour week) and move up in Table 1. We believe that such mapping assists the organization of what follows

in the process, since it introduces the complexity in stages: First, the simple-to-implement practices are introduced, and later on the more complicated practices are addressed. In this paper we have restricted our analysis to the two chasms described above, and do not discuss other benefits of XP discussed in the literature (for instance, in [1]).

40-Hour Week. As described in XP books, the purpose of this practice is to maintain freshness. Interestingly, even this simple-to-implement practice contributes to the reduction in the cognitive complexity involved in software development. The work framework that this practice establishes, in which programmers develop software for only 8 hours per day, enables programmers to detach themselves from the details involved in software development and to take a more distanced look at the software after they finish the actual 8-hour development process. In contrast, when software developers spend 12 (and more) hours each day engaged in software development activities, they have no opportunity to reflect on the software from the "outside". This practice, therefore, helps one examine the developed software without being overwhelmed by details. In other words, the 40-hour week practice may enable the programmer, in many cases unconsciously, to consider the software development in terms of higher levels of abstraction, i.e. to bridge the abstraction chasm. We believe that the ability to view the system on a higher level of abstraction improves the code that is to be developed the following day.

In addition, this practice helps bridge the satisfaction chasm as well. The 40-hour week practice allows the programmer to dedicate more time to his or her personal life without having to fulfill other people's expectations by working long hours. In other words, this practice helps one decide whose satisfaction and expectations to fulfill. Since the entire team works according to this practice, it is assured that no conflict will exist that must be overcome.

On-Site Customer. The fact that a real customer is part of the team and is available full-time to answer questions, enables developers to avoid making decisions regarding the customer's needs without checking first with the customer what is really needed and whether the decisions made are correct. That is, in addition to the importance of this practice from the customer's perspective, the immediate feedback that the on-site customer gives the developers helps them overcome a tendency to make decisions on their own (without the customer) in order to fulfill their need to proceed with the development process. In this way, this practice helps bridge the satisfaction chasm.

Small Releases. This practice promotes transitions between levels of abstraction. In a release planning, a global view is obtained. Details are added only with respect to the customer stories that are intended to be implemented in the forthcoming release. The recommended time period that leads to small releases (every 4-6 months) enables the programmer to move between levels of abstraction without spending too much time in abstraction levels that are too high (requirement analysis) or too low (coding).

Continuous Integration. This practice as well promotes transitions between levels of abstraction. In this case, the movement between abstraction lev-

els depends on the results of the tests. If all the tests are passed together with the integrated code, the developer can remain on the same level of abstraction throughout the coding phase, since it is appropriate to stay on that level of abstraction. If the tests fail, the developer must gain a more global view of the system. He or she has to examine the application on a higher level of abstraction in order to find the source of the problem and the way in which the integrated code affects the rest of the code, as well as to consider possible solutions.

Pair Programming. This practice specifies that any code segment must be written by two developers, each of which has a different role. The developer with the keyboard and mouse must consider what the best way to implement a specific task is, while the other partner must think in a more strategic manner. It is easy to see how this practice can help bridge the abstraction chasm. Since the two individuals work on different levels of abstraction - the driver thinks locally; the navigator thinks globally - the same task is considered on two different levels of abstraction *at the same time*.

However, pair programming contributes also to the bridging of the satisfaction chasm. In this case, the chasm is represented by the gap between the individual tendency to move on as soon as the code is running (sometimes without testing it or rendering it more readable), and the benefits that the collective can gain from this practice, i.e. a code that is comprehensible by a greater number of people. In addition, pair programming also helps to stay focused on the task, while the individual tendency may be to engage in other activities (email, for instance), to skip testing, and so on.

Planning Game. In addition to its significant role in meeting the customer needs, the planning game, like the small releases practice, leads software developers to move between different levels of abstraction several times within the course of the development process. The release planning game is carried out on high levels of abstraction, while the iteration planning game takes place on lower levels of abstraction. Further details (such as development tasks and time estimations for the development tasks) are addressed only with respect to the iteration under discussion. In addition, the planning game enables the developers to see the way in which the system is composed of its components. Being part of the entire process, one can see the entire picture and at the same time observe the process through which the system is built up from its parts. This stands in contrast to a situation in which programmers are required to implement components that have been planned or designed by others. All of the above illustrates how the planning game is one of the XP practices that encourage developers to move between levels of abstraction - a transition which is inherent in software development processes and which is difficult to apply in the absence of specific practices that lead such a process - thus helping to bridge the abstraction chasm.

Coding Standards. Since programmers must write all code in accordance with rules that emphasize communication through the code, they must curb their tendency to write code in a way that is comprehensible to them alone. In other words, the coding standards practice leads one to overcome his or her individual

needs in order to fulfill collective needs, thereby supporting the bridging of the satisfaction chasm.

Testing. In addition to the huge contribution of this practice to the correctness of software, this practice (with its strict and rigid implementation in XP) helps the individual bridge the satisfaction chasm. Testing helps the programmer avoid perceiving any code that runs as a complete code and prevents him or her from leaving any piece of code before all tests, written for it in advance, are passed successfully.

Furthermore, by the implementation of the XP practice of testing coding becomes a series of manageable steps through which small pieces of code are added to the application. Thus, testing helps one remain on a low level of abstraction when appropriate. This approach stands in contrast to situations in which the process of testing requires the mental manipulation of large amounts of code, a process that requires simultaneous consideration of the code on different levels of abstraction. This is of course very difficult and may explain the negative feelings that people associate with software testing. Not surprisingly, in XP development frameworks these negative feelings towards testing are transformed into more positive feelings.

Metaphor. In his talk *The Metaphor Metaphor*, given in OOPSLA 2002, Kent Beck said that "[o]f all the aspects of Extreme Programming, the suggestion that customers and developers share a common metaphor or metaphors for the system is the most problematic." We would like to present this practice as a means by which the bridging of the satisfaction chasm is assisted. Specifically, this practice encourages developers to look at the other person's perspective (customer, teammate) while engaged in development of the software. Instead of being "trapped" in one's own conception of the software development process, programmers must also take into consideration the perspective of the others in order to help everyone involved in the development process understand the elements of the software and the relationships between them. The chasm is especially noticeable when dealing with customers who are not always familiar with the technical jargon used by software developers.

Collective Ownership. As it turns out, since programmers know that their work will be examined by others and improved upon if required, there is a tendency to postpone immediate personal satisfaction and improve the code prior to its integration. Thus, the bridging of the satisfaction chasm is supported. In addition, when refactoring codes written by other developers, one must adopt the other's perspective in order to enhance his or her own understanding of the code.

Refactoring. In addition to the fact that refactoring improves communication and code simplicity, it also helps bridge the abstraction chasm throughout coding phases. This can be simply observed from the fact that in order to improve a piece of code one must examine it on a level of abstraction that is higher than that in which it was written.

Furthermore, refactoring also helps bridge the satisfaction chasm. According to this practice, it is not sufficient that the code pass all the tests. The code must

also be restructured and improved upon in order to enjoy the benefits of long-term activities that are to be performed on the code, either by the developer who originally wrote the code or by other developers. In other words, the investment of a greater deal of effort now (that is, postponing individual's current desire to move on), will save time and effort in the future.

Simple Design. Like refactoring, this practice also aims at improving communications between the teammates working on the same software. This practice, too, promotes transition between different levels of abstraction. This is achieved by examining the code (low level of abstraction) by a higher level of abstraction (the simple design). Furthermore, the practice of simple design also helps to bridge the satisfaction chasm. It is not sufficient that the code pass all the tests, its design must also be simplified as much as possible.

4 Conclusion

This paper examines how XP practices support the bridging of two chasms that exist in software development - the abstraction chasm and the satisfaction chasm. In the case of the abstraction chasm, XP practices guide software developers to move between different levels of abstraction, or alternatively to stay at the same level of abstraction, all according to the case. In the case of the satisfaction chasm, XP practices guide software developers to postpone satisfaction of their own needs and, when appropriate, to invest more efforts in fulfilling the needs and expectations of others. We suggest that this observation can explain the rapid acceptance of XP by the industry. In our current work we expand this discussion and illustrate additional ways by which XP practices can reduce cognitive and social complexities of software development.

References

1. Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley 2000
2. Hazzan, O. and Dubinsky, Y.: Teaching a software development methodology: The case of Extreme Programming. In *The proceedings of the 16th International Conference on Software Engineering Education and Training*. Madrid, Spain 2003