Reflective practitioners in software design. Case study :Extreme programming

Cestmir Halbich, Department of information technologies, CUA Prague

Abstract: The position paper comes out from Donald Schoen's ideas about reflective practitioners and describes author's experience in area of computer science. The software gap is mentioned and methods for its overbridging too. At the case study is illustrated Schoen's approach to designer's activity and benefit for improving effectiveness and efficiency of the software design process by reflective practitioner's approach. In the conclusion the advantages and disadvantages are described.

Key words: reflective practitioner, extreme programming, software gap

By Schoen designers use design representations to create a virtual "design world" [Schoen 1992] in which the objects and relationships of the design situation are named and made explicit. The situation that the designer constructs with representations enables new understandings of the design problem by allowing the designer to see ideas that before existed only tacitly, and to understand implications of the design situation that did not exist before constructing the representations. In accordance with some authors we can say that putting ideas down on paper is not a matter of emptying out the mind but of actively reconstructing it, then forming new associations, and expressing concepts in pictorial, linguistic, or any explicit representational forms. Designers work mainly with matter, software developers work usually with algorithms, but the analogy is evident.

Knowledge is constructed in programming through an interaction between the programmer's understanding of the design situation and the representations the programmer's creates. Design theorist Donald Schoen characterises the relationship between the designer and design representations as a "reflective conversation with the materials of the situation" [Schoen 1983], in which the designer acts to shape the design situation by creating or modifying design representations, and the situation talks back to the designer, revealing unanticipated consequences of the design actions. By analogy the programmer reflects on the actions and consequences to understand the situation's back-talk, and then plans the next course of action. The software design process is driven by the interaction between the programmer and design algorithms, rather than by following a pre-planned solution approach (see Figure 1).

Knowledge is constructed in software design through repeated cycles of representing and interpreting the design situation. Programmers construct the design situation that talks back in the form of a breakdown. This back talk is interpreted, or reflected upon, to activate

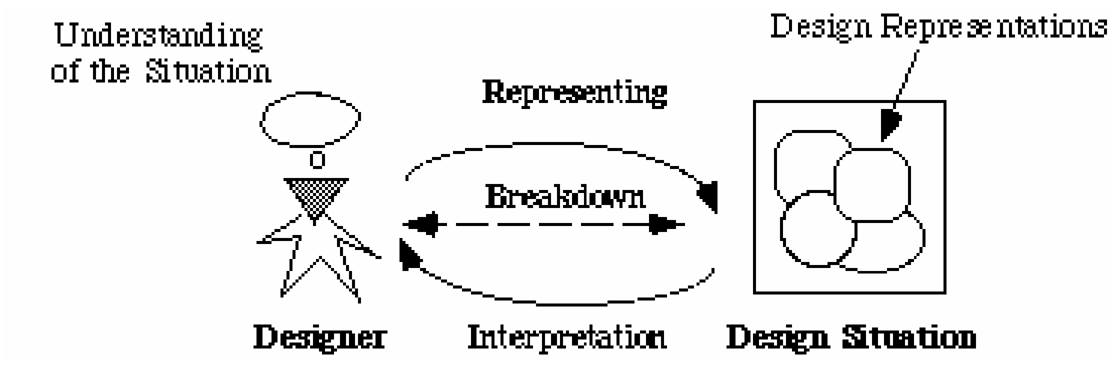

Figure 1. Knowledge Construction in software Design

new understandings of the design situation. In this situation is very useful to use certain formal methodology to improve the effectiveness and efficiency of the software design process.

Already at the conference NATO 1968 the existence of the gap- “gap, between what was hoped for from a complex software systems, and what was typically achieved.” There is a widening gap between ambitions and achievements in software engineering. “This gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious.“ [NATO 1968, p.70]. Since this time amount of different methodologies was invented. One of these is so called extreme programming [Beck 1999].

By author’s opinion the extreme programming is very closely joined with Donald Schoen’s ideas about reflective practitioners. Contrary to some other programmer’s techniques extreme programming closely collaborates with users in the all phases of software design. Some features of extreme programming are discussed below.

Extreme Programming (XP) is actually a deliberate and disciplined approach to software development. It has already been proven at many companies of all different sizes and industries world wide. XP is successful because it stresses customer satisfaction. The methodology is designed to deliver the software your customer needs when it is needed. XP empowers your developers to confidently respond to changing customer requirements, even late in the life cycle. This methodology also emphasises team work. Managers, customers, and developers are all part of a team dedicated to delivering quality software. XP implements a simple, yet effective way to enable groupware style development. XP improves a software project in four essential ways; communication, simplicity, feedback, and courage. XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested.

Now we in short describe main features of all phases of software design by extreme programming.

Features in Planning

User stories are written. Release planning creates the schedule. Make frequent small releases. The project velocity is measured. The project is divided into iterations. Iteration planning starts each iteration. Move people around. A stand-up meeting starts each day. Fix XP when it breaks.

Features in Designing

Simplicity. Choose a system metaphor. Use CRC-cards for design sessions. Create spike solutions to reduce risk. No functionality is added early. Refactor whenever and wherever possible.

Features in Coding

The customer is always available. Code must be written to agreed standards. Code the unit test first. All production code is pair programmed. Only one pair integrates code at a time. Integrate often. Use collective code ownership. Leave optimisation till last. No overtime.

Features of Testing

All code must have unit tests. All code must pass all unit tests before it can be released. When a bug is found tests are created. Acceptance tests are run often and the score is published.

Now we describe some details about pair programming. All code to be included in a production release is created by two people working together at a single computer. Pair programming increases software quality without impacting time to deliver. It is counter intuitive, but two people working at a single computer will add as much functionality as two working separately except that it will be much higher in quality. With increased quality comes big savings later in the project. The best way to pair program is to just sit side by side in front of the monitor. Slide the key board and mouse back and forth. One person types and thinks tactically about the method being created, while the other thinks strategically about how that method fits into the class. It takes time to get used to pair programming so don't worry if it feels awkward at first.

Conclusion

We have good personal experience with implementing of the extreme programming in the practice. Reflective practitioner method improves added value of the collaboration in software design process by iterations and feedback loops with customers and end users (see Figure 2). The case study's Czech company has good experience with software design of small, medium and large information systems and work with small and medium work teams.

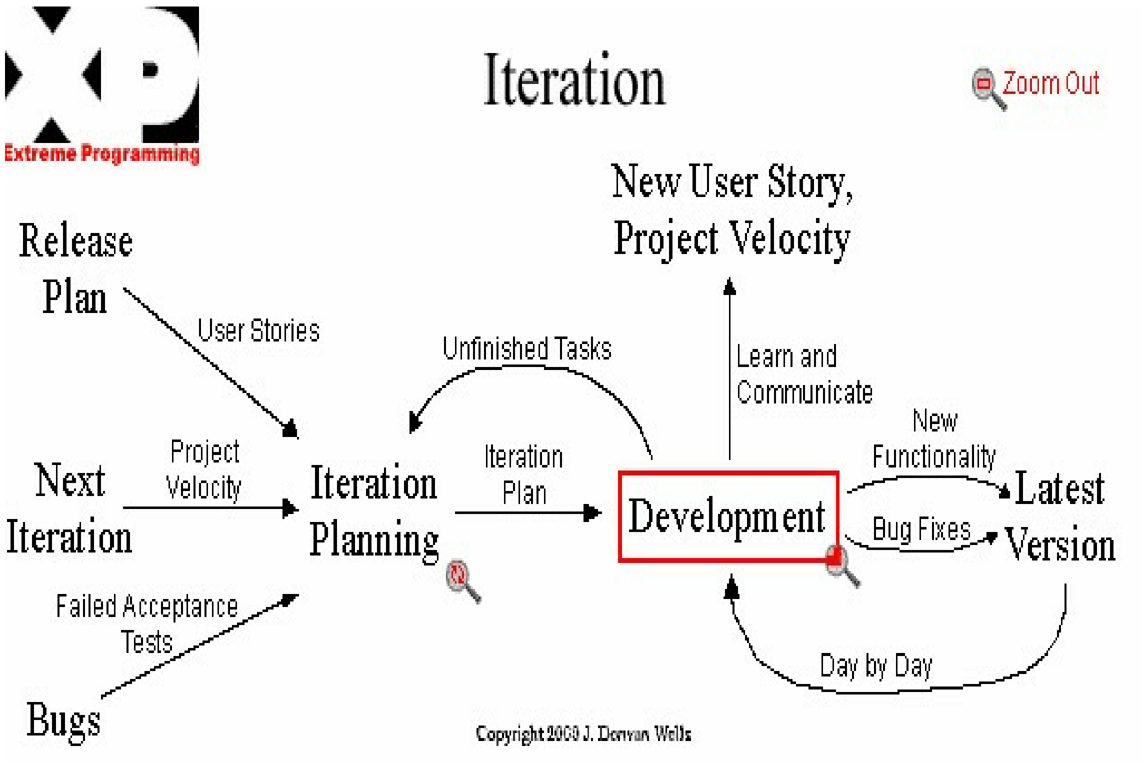

Figure 2. Iterations and feedback lops in extreme programming

References

NATO SOFTWARE ENGINEERING CONFERENCE 1968, Garmisch, 1968

SCHOEN, D.A. The Reflective Practitioner: How Professionals Think in Action, Basic Books, New York, 1983, ISBN: 0465068782

SCHOEN, D. "Designing as reflective conversation with the materials of a design situation," Knowledge-Based Systems Journal - Special Issue on AI in Design, Vol. 5, No. 1, 1992, pp. 3-14

BECK, Kent: Extreme Programming Explained: Embrace Change, 224 pages ; Addison-Wesley Pub Co; 1st edition,1999, ISBN: 0201616416