In G. Kadoda (Ed). Proc. PPIG 13 Pages 237-248

The Application of Reflective Practitioner Perspective to Software Engineering

Orit Hazzan
Department of Education in Technology & Science
Technion – Israel Institute of Technology
Haifa, Israel 32000
oritha@techunix.technion.ac.il
Fax: (9724) 832 5445

Keywords: POP-I.A. *training design* POP-II.B. *design* POP-V.B. *protocol analysis*

> The process of preparing programs for a digital computer is especially attractive because it not only can be economically and scientifically rewarding, it can also be an aesthetic experience much like composing poetry or music. (Knuth, 1969, p. V)

Introduction

This paper focuses on the application of the Reflective Practitioner (RP) perspective to the discipline of Software Engineering (SE). The Reflective Practitioner perspective, introduced by Donald Schön (1983, 1987), guides professional people (architects, managers, musicians, and others) to rethink and examine their professional creations during and after the accomplishment of the creation process[1]. The working assumption is that such a reflection improves the proficiency and performance within these professions. An analysis of the field of Software Engineering in general, of its sub-field Software Architecture[2] in particular, and of the kind of work that software engineers usually carry out, all support the adoption of the RP methodology to SE.

In an ongoing research I collect data about student reflection on the process of developing a software system. For reasons of space limitation I will not present the data itself in this paper. In the talk I will illustrate the ideas presented in the paper by quoting student reflection. This illustration may also shed light on the adopting the RP methodology for educational purposes of SE. Further support for this approach can be found in Tomayko (1996) and Wallingford (1998).

Why the RP paradigm should be applied to SE

> *Our question then is not so much whether to reflect as what kind of reflection is most likely to help us get unstuck.* (Schön, 1983, p. 280)

In his two books which address the RP framework and RP education (Schön 1983, 1987), Schön analyses the contribution one may receive from continuously thinking, examining, reformulating and re-expressing one's practice and one's thinking about his/her practice. With respect to science and engineering Schön says that "[b]etween 1963 and 1982 … [i]ncreasingly we have become aware of the importance to actual practice of phenomena – complexity, uncertainty, instability, uniqueness, and value-conflict". (Schön, 1983, p. 39). In that period of time, the computing community observed a similar phenomenon with respect to developing software systems. It was in the fall of 1968, when the Software Crisis term was introduced in the first-ever NATO Conference on Software Engineering, in Garmish, Germany. As a result, it was recognized that software development should be guided by a professional-systematic approach. The mental complexity involved in such projects was acknowledged, and, as a result, there was tremendous awareness of the impossibility of managing software systems without systematic (engineering oriented) methods. However, though the art (or profession) of programming, together with its complex nature were known at the time when Schön wrote his books, he did not discuss the application of RP with respect to SE.

---

[1] These activities are called reflection-in-action and reflection-on-action respectively.

[2] Cf. Worldwide Institute of Software Architects (http://www.wwisa.org/wwisamain/index.htm). In addition, look at http://www.sei.cmu.edu/architecture/definitions.html for a list of definitions of software architecture.

The rest of this section is organized around 7 quotes from the two RP books. The relevance of these topics to SE is examined, and they are discussed from professional, cognitive and educational points of view.

A professional knowledge:

> *As Edgar Schein has put it, there are three components to professional knowledge:*
>
> 1. *An underlying discipline or basic science component upon which the practice rests or from which it is developed.*
> 2. *An applied science or "engineering" component from which many of the day-to-day diagnostic procedures and problem solutions are derived.*
> 3. *A skills and attitudinal component that concerns the actual performance of services to the client, using the underlying basic and applied knowledge.* (Schön, 1983, p. 24)

The ongoing discussion on the nature of the discipline of Computing Science is well known (CF. The ACM/IEEE-CS Joint Curriculum Task Force Report, Turner, 1991). In that document, nine subjects[3] are presented as the subject areas that are identified as comprising the core of the discipline. An examination of these nine areas reveals that we can find the above three components in Computing Science. More specifically, among these nine areas there are elements of basic science (e.g., algorithms and data structures), of "engineering" (e.g., software methodology and engineering) and of "skills and attitudinal component" (e.g., human-computer interaction). Thus, in fact, the basis for continuing the application of the RP perspective to SE is established.

Teaching professional methodologies:

> *Professionals have been disturbed to find that they cannot account for processes they have come to see as central to professional competence. It is difficult for them to imagine how to describe and teach what might be meant by making sense of uncertainty, performing artistically, setting problems, and choosing among competing professional paradigms, when these processes seem mysterious in the light of the prevailing model of professional knowledge.* (Schön, 1983, page 19-20)

Though there are many discussions within the community of Computing Science about the importance of Computing Science methodologies, and the importance of these methodologies is acknowledged in programming courses, it is not trivial to teach the essence of these methodologies. This phenomenon is partially a result of the fact that the understanding of these methodologies should be based on one's personal experience and reflection on one's creation. This point of view is discussed in *Educating the Reflective Practitioner* (Schön, 1987). Schön bases his theory on Dewey, saying:

> *The student cannot be taught what he needs to know, but he can be coached: "He has to see on his own behalf in his own way the relations between means and methods employed and result achieved. Nobody else can see for him, and he can't see just by being 'told', although the right kind of telling may guide his seeing and thus help him see what he needs to see."* (Dewey, 1974, p. 151). (In Schön, 1987, Page 17)

Thinking about the RP methodology as a cognitive tool, it may help programmers in developing computer programs in general, and may add to student understanding of Computing Science methodologies in particular. The reason is that RP aims to improve one's understanding of the process of creating complex objects. At the same time one may deepen one's understanding of the principles and guidelines that lead the creation process. More specifically, in the context of SE, the better one understands the process of developing a software system, the better one may understand the methodologies that guide this process. By adopting the RP methodology to SE education, student ongoing reflection of the process of developing software systems becomes part of the programming process and thus, they may improve their understanding of Computing Science methodologies.

[3] The nine subjects are: algorithms and data structures; architecture; artificial intelligence and robotics; database and information retrieval; human-computer interaction; numerical and symbolic computation; operation systems; programming languages; software methodology and engineering.

The engineering schools education:

> *In his 1967 article, "Dilemmas of Engineering Education", he [Harvey Brooks, the dean of Harvard Engineering Program] described the predicament of the practicing engineer who is expected to bridge the gap between a rapidly changing body of knowledge and the rapidly changing expectations of society. The resulting demand for adaptability, Brooks thought, required an art of engineering. But the scientizing of the engineering schools had been intended to move engineering from art to science.* (Schön, 1983, p. 171)

Many papers currently deal with the orientation of engineering schools and address dilemmas in engineering education. For example, Glass (1977) points at the industry/academe communication chasm, saying that "[a]cademic people tend to assume that student problems are typical programming problems, and that the real-world can be simulated in one (or two) semester projects. Industrial people tend to reinvent the same ad hoc wheel they invented last year, and not even remove any of the flat spots". (p. 13).

Inside the community of Computing Science there are those who voice the opinion that the way computer programs are presented to students in university courses does not reflect the actual complexity involved in the process of developing computer programs. In fact, there are two aspects of that complexity. The first aspect refers to the complexity of the actual process of developing software systems; the second aspect refers to the complexity of the social environment in which software systems are developed, which includes for example, customers and team-members. Denning (1992) says with respect to this social complexity that *"[e]mployees and business executives complain that graduates lack practical competence. Graduates, they say, cannot build useful systems, formulate or defend a proposal, write memos, draft a simple project budget, prepare an agenda for a meeting, work on teams, or bounce back from adversity; graduates lack a passion for learning"* (p. 84).

Denning (ibid.) points to the way in which new engineers are educated as a source of this problem. He says: *"Our approach to education, in which traditional engineering education is rooted, is founded on an unspoken assumption that before we can take effective action, we must have an accurate model of the world, which we gain by acquiring knowledge. Consequently, our teaching is organized as a continuing presentation of important facts, procedures, methods, and models, transferring to our students a subset of body of knowledge constituting the discipline. Our curricula are specifications of theses presentations. […] [W]e should recognize a second kind of knowledge besides facts, procedures, rules, and models – the kind of knowledge that can be gained only from involvement with others in the community who already know it. The second kind of knowledge includes knowing how to listen, to design, to care, to persuade, to be organized for new learning, to be a professional, and even to be trustworthy and honest."* (p. 86). These topics are, in fact, similar to the ideas that the RP framework suggests to addressing.

SE graduates express a similar conception of SE education. Lethbridge (2000) presents the result of a survey of software practitioners conducted during the summer and autumn of 1998. The survey was designed to ascertain what knowledge is important to the participants and, to better understand participant educational and training needs. Among other conclusions, the following finding is relevant to our discussion. The finding presents a list of topics for which, according to the participants, there is the greater necessity for improvement in university courses. The first seven topics in this list are: software design and patterns; requirements gathering and analysis; software architecture; human-computer interaction/user interfaces; object-oriented concepts and technology; ethics and professionalism; analysis and design methods (p. 66). Examining the nature of these topics, we may suggest that the application of RP into the Computing Science curriculum may improve student understanding of these subjects. What I suggest is that even though RP does not directly refer to these skills, learning to be a RP may help in acquiring mental skills needed for coping with these topics.

The reflection subjects:

> *When a practitioner reflects in and on his [her][4] practice, the possible objects of his reflection are as varied as the kinds of phenomena before him and the systems of knowing-in-practice which he brings to them. He may reflect on the tacit norms and appreciations which underlie a judgment, or on the strategies and theories implicit in a pattern of behavior. He may reflect on the feeling for a situation which has led him to adopt a particular course of action, on the way in which he has framed the problem he is trying to solve, or on the role he has constructed for himself within a larger institutional context.* (Schön, 1983, p. 62)

Laying out the topics which are possible subjects for reflection in SE, we may start with the actual creations (the software systems themselves), going through a reflection on the way algorithms are used in the software systems, and moving to softer topics such as development approaches, ways of thinking, etc. As I see it, based on the assumption that the RP methodology is suitable for SE, the topics that should be the main subjects of the reflection deserve further study and elaboration.

Conversation with the material:

> *In the designer's conversation with the material of his design, he can never make a move which has only the effects intended for it. His materials are continually talking back to him, causing him to apprehend unanticipated problems and potentials. As he appreciates such new and unexpected phenomena, he also evaluates the moves that have created them.* (Schön, 1983, p. 100-101)

The analogy in this case seems to be trivial. When one develops a software system, one actually is in an ongoing conversation with the creation. In fact, parts of the software systems are shaped in an ongoing interaction with the computer as a mediator that reflects to the software designer how far away he/she is from what s/he wants to achieve. In other words, the computer is the medium through which a software designer talks to his/her creation – the software system.

Stepwise refinements:

> *Neither practitioner can know, at the moment of reframing, what the solution to the problem will be, nor can he be sure that the new problem will be soluble at all. But the frame he has imposed on the situation is one that leads itself to a method of inquiry in which he has confidence.* (Schön 1983, p. 134)

Stepwise refinements is one methodology used in developing software systems for overcoming the complexity involved in developing such systems. As the above quote suggests, the methodology of successive refinements implies that one may consider that what one wants to be programmable can be programmed. In that spirit, successive, formal refinement is defined as "*[m]ethodology for resolving discrepancies between the host language and target models*"[5]. Wirth (1971) says that in the process of stepwise refinements *"[i]n each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. […] Every refinement step implies some design decisions. It is important that these decisions be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions."*

Once again, it is easy to bring to light the similarities between the assumption on which the RP paradigm has been developed and the foundations of Computing Science methodologies. These similarities suggest the adaptation of RP into SE in general and into SE education in particular.

Science vs. engineering in RP professions:

> *[A]ccording to the model of Technical Rationality, emphasis is placed on the separation of research from practice. On this view, practice should be based on scientific theory achievable only through*

[4] For the reader convenience, I add the feminine pronoun only once. However, this should be applied in all other quotes.
[5] http://www-cad.eecs.berkeley.edu/~jimy/research/cadlunch.4.98/tsld009.htm (Shin Young)

> *controlled experiment, which cannot be conducted rigorously in practice. So to researchers and the research setting falls the development of basic and applied science, while to practitioners and the practice falls the use of scientific theories to achieve the instrumental goals of practice.* (Schön, 1983, p. 144-145)

Indeed, "[c]omputer science is neither a science nor is it engineering in a traditional sense. Computer science defined a new relation between theory and experiments. The science and engineering aspects in computer science are much closer than in many other disciplines and that is what makes computer science unique." (El-Kadi, 1999, p. 27.)

One topic in Computing Science which demonstrates this relation between theory and practice, is correctness of computer programs. On the one hand, it is accepted that it is almost impossible to prove the correctness of a program with more than 30 LOCs (or so). On the other hand, the theory of proof of correctness enables proving the correctness of some length of LOC. A similar phenomenon can be observed in professions for which Schön suggests adopting the RP methodology.

How can the RP perspective be integrated into SE education

In the spirit of *Educating the Reflective Practitioner* (Schön, 1987), by which "education for reflective practice, thought not a sufficient condition for wise or moral practice, is certainly a necessary one" (p. xiii), one of the tasks I proposed to students in a course about the human aspect of SE (cf. Hazzan, 2001), was to conduct an ongoing reflection on the process of developing a programming project. This was a local attempt to educate the students toward a reflective mode of thinking.

The course was conducted as a workshop including discussion, reflection, teamwork, and feedback. The students had to develop a project which aims at operating an airport within the limitation of the semester framework. The project was developed in teams where each team developed one part of the system. The programming aspect of the task was open to student decision. The students could determine the scale of the project and what programming language to use. The idea was to let them struggle with problems arising out of these issues so that they would have to consider various possibilities.

At the end of the project the students had to submit a written report emphasizing computing issues as well as reflection on the development process. This provided another opportunity for the students to examine and rethink the process they had gone through.

In the spring semester of 2001 I will implement a programming studio similarly to the architecture studio (cf. Tomayko, 1996; Wallingford, 1998). In the studio, students develop a software system while maintaining an ongoing reflection of what has been learned (both on the personal level and on the team level). The spirit of the studio is inspired by the traditional studio of architecture studies:

> *Studios are typically organized around manageable projects of design, individually or collectively undertaken, more or less closely patterned on projects drawn from actual practice. They have evolved their own rituals, such as master demonstration, design review, desk crits, and design juries, all attached to a core process of learning by doing. And because studio instructors must try to make their approaches to design understandable to their students, the studio offers privileged access to designers' reflections on designing. It is at once a living and traditional example of a reflective practicum.* (Schön, 1987, p. 43)

As in the architecture studio, in the programming studio too, several kinds of learning may be intertwined:

> *In this process, several kinds of learning are interwoven. The student learns to recognize and appreciate the qualities of good design and competent designing, in the same process by which she also learns to produce these qualities. She learns the meanings of technical operations in the same process by which she learns to carry them out. And as she learns to design, she also learns to learn to design – that is, she learns the practice of the practicum.* (Schön, 1987, p. 102).

The programming studio is accompanied with a research on programming methods, ways of thinking, programming discourse, and social processes.

Epilogue

This paper outlines some thoughts about the application of the Reflective Practitioner perspective into the community of software engineers/designers. I would like to end with the following quote, in the hope that the RP perspective is not misinterpreted:

> *When a practitioner keeps inquiry moving, however, he [she] does not abstain from action in order to sink into endless thought. Continuity of inquiry entails a continual interweaving of thinking and doing.* (Schön, 1983, p. 280).

References

Dewey, J. (1974). *John Dewey on Education: Selected Writings.* (R. D. Archambault, ed.). Chicago: University of Chicago Press.

Denning, P. J. (1992). Educating a new engineer, *Communication of the ACM* 35(12), pp. 82-97.

Duhalbom, B and Mathiassenm L. (1997). The future of our profession, *Communication of the ACM* 40(6), pp. 80-89.

El-Kadi, A. (1999). Stop that divorce! *Communication of the ACM* 42(12), pp. 27-28.

Glass, R. L. (1997). Revisiting the industry/academe communication chasm, *Communication of the ACM* 40(7), pp. 11-13.

Hazzan, O. (In press, 2001). Teaching the human aspect of software engineering - A case study, *Proceeding of SIGCSE 2001 - The 32nd Technical Symposium on Computer Science Education*, Charlotte, NC, USA.

Knuth, D. E. (1969, 2nd Printing). *The Art of computer Programming,* Addison-Wesley Pubkishing Company.

Lethbridge, T. C. (2000). Priorities for the Education and Training of Software Engineers, The *Journal of Systems and Software*, 53(15), pp. 53-71.

Schön, D. A. (1983). *The Reflective Practitioner*, BasicBooks,

Schön, D. A. (1987). *Educating the Reflective Practitioner: Towards a New Design for Teaching and Learning in The Profession*, San Francisco: Jossey-Bass.

Tomayko, J. E. (1996). Carnegie-Mellon's Software Development Studio: A Five-Year Retrospective, *SEI Conference on Software Engineering Education*.

Turner, A. J. (1991). The ACM/IEEE-CS Joint Curriculum Task Force Report, *Communications of the ACM* 34(6), pp. 69-84.

Wallingford, E. (1998).). Software studio, http://www.cs.uni.edu/~wallingf/teaching/162/studio.html.

Wirth, N. (1971). Program Development by Stepwise Refinement, *Communications of the ACM* 14(4), pp. 221-227. http://www.acm.org/classics/dec95/.