

# The Reflective Practitioner Perspective in eXtreme Programming

Orit Hazzan<sup>1</sup> and Jim Tomayko<sup>2</sup>

<sup>1</sup>Department of Education in Technology and Science, Technion - IIT, Haifa 32000, Israel  
oritha@tx.technion.ac.il

<sup>2</sup>School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, U.S.A.  
jet@cs.cmu.edu

## ABSTRACT

This paper examines ways by which a reflective mode of thinking may improve eXtreme Programming (XP) practices. It describes the reflective practitioner perspective and suggests specific ways in which such an approach may be interwoven into XP practices. Specifically, the focus is placed on the construction of ladders of reflection. These ladders illustrate how one may increase the level of abstraction of his/her thinking when reflection is interwoven in the process of software development, and how such an experience may promote one's comprehension of the relevant development process.

## Keywords

reflection, extreme programming, reflective practice, software engineering.

## 1 Introduction

This essay suggests adding a reflective practice (RP) perspective to eXtreme Programming (XP). Based on Donald Schön's work with educating professionals, it is suggested that as a reflective practitioner one may improve the performance of the XP practices. Generally speaking, the RP perspective, first introduced by Donald Schön [11, 12], guides professional practitioners (such as architects, managers, musicians and others) towards examining and rethinking their professional creations during and after the accomplishment of the process of creation. The working assumption is that such a reflection improves both proficiency and performance within such professions. Analysis of the field of Software Engineering (SE) and the kind of work that software engineers usually accomplish in general [10], and the XP practices in particular, support the adoption of the RP perspective to SE in general and to XP in particular. Specifically, it is suggested that a reflective mode of thinking may improve the application of some of the XP practices. This paper examines this possible contribution.

We present the main ideas behind the RP perspective, discuss the potential contribution of the RP perspective to XP, and, based on this analysis, suggest directions for future research.

## 2 Reflective Practice<sup>1</sup>

The two main books which present the Reflective Practice (RP) perspective are *The Reflective Practitioner* [11] and *Educating the Reflective Practitioner* [12]. While the first book presents professions for which reflective thinking is (or should be) inherent in, such as architecture and management, the second book focuses on how *to educate* students of such professions to be reflective practitioners. In this section we establish the rationale for implementing the RP perspective to SE in general and to XP in particular.

In the two books mentioned above, Schön analyses the added advantages one may obtain from continuously examining one's practice and one's thinking about his/her practice. With respect to science

---

<sup>1</sup> This section is largely based on Hazzan ([5]).

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”. ([11], p. 39). At that time, the Computer Science community observed a similar phenomenon with respect to developing software systems (Cf. the “Software Crisis” terminology introduced in 1968 at the NATO Conference in Garmish, Germany). Many at the conference recognized that software development should be guided by a professional-systematic approach. The mental complexity involved in developing software 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 complex nature of the profession of software development was known at the time when Schön wrote his books, he did not discuss the application of the RP perspective with respect to SE. Yet, the relevance of RP to SE in general and to XP in particular, is illustrated by the following three quotes taken from Schön’s work.

#### **The subjects of reflection:**

*When a practitioner reflects in and on his 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. ([11], p. 62)*

Laying out the topics which are possible subjects for reflection in SE, we may start with the actual creations (the software systems), going through a reflection on the way algorithms are developed and used in software systems, and moving on to skill-related topics such as development approaches, topics related to human-computer interaction, aspects of software development methodologies, ways of thinking, etc. In fact, it seems that we might end up with a rich object collection that can be subjects of thought. It might be the result of the fact that “[m]any of the things we make with software today are more complex than most buildings and, as in building design, software design embraces many aspects: function, safety, human interface, ergonomics, graphics, algorithms, data structure, program structure, protocol, and application interface, among others.” [14]. If we limit the discussion to XP practices, we may suggest the following objects for reflection: the way unit-tests are developed, how a specific simple design was determined, how a specific path of refactoring emerged, etc.

#### **Listening to the code:**

*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. ([11], p. 100-101)*

The analogy to SE with respect to this topic seems to be trivial [10]. When one develops a software system, one actually is in an ongoing conversation with the creation. In fact, several aspects of software systems are shaped in an ongoing interaction with the computer as a mediator that reflects to the software developer how far away s/he is from what s/he wants to achieve. In other words, the computer is the medium through which a software constructor talks to his/her creation – the software system. Within the XP framework, this kind of interaction with the material is especially dominant in the process of unit-testing and in refactoring processes that interweave on-going testing.

#### **The ladder of reflection:**

*We can [...] introduce another dimension of analysis [for the chain of reciprocal actions and reflections that make up the dialogue of student and coach in the architecture studio]. We can begin with a straightforward map of interventions and responses, a vertical dimension according to which higher levels of activity are “meta” to those below. To move “up”, in this sense, is to move from an activity to reflection on that activity; to move “down” is to move from reflection to an action that enacts reflection. The levels of action and reflection on action can be seen as the rungs of a ladder. Climbing up the ladder, one makes what has happened at the rung below an object of reflection. ([12], p.114)*

The ladder of reflection described in this quote refers to student-tutor dialogue in the architecture studio. Hazzan ([5]) expands the ladder of reflection presented by Schön to a student-coach dialogue in a software studio and with respect to an individual work. The idea in both cases is to illustrate how one may increase the level of abstraction of his/her thinking when reflection is interwoven in the process of software development. In the continuation of this paper a ladder of reflection is presented with respect to a pair programming session, a planning game session and a refactoring process. These cases come to illustrate how such a ladder of reflection may promote one's comprehension of the relevant development process and may lead to insights that eventually may save time and money.

### 3 Reflective Practice in eXtreme Programming Practices

This section illustrates how a RP perspective may support and improve the application of the XP practices. First we explain the fitness of RP to XP. Then, the potential contribution of the RP perspective to each of the XP practices is examined.

It seems that a RP approach fits very well to XP, since XP emphasizes learning through reflection processes. For example, the estimation of the team's velocity is improved from project to project based on a reflective process; when a pair is engaged in a pair programming session, the navigator reflects on the drivers' coding. Thus, it seems that one of the implicit XP guidelines is reflection. Still, as far as we know, it is not outlined inherently in the practices themselves. Similarly to some of the XP practices, RP is not explicitly directed to code production but in the long term it may improve code production and quality. As XP incorporates activities that are not directly oriented to code production, yet may improve code development processes, we suggest that the RP perspective may be integrated naturally in XP.

This work follows other publications that emphasize the importance of reflection and retrospective in the context of software development in general and with respect to agile methods in particular (such as [6] and [2] respectively). We propose that our contribution is in the introduction of a reflective perspective into each of the XP practices by the construction of ladders of reflection that guide software developers to think on higher levels of abstraction.

In the discussion that follows, the XP practices are gathered in three groups, according to the subject they focus on: the team, the customer and the code. We do not claim that each practice focuses only on one of these three subjects. However, it seems that each of the XP practices influences significantly one subject out of these three.

Team	Customer	Code
Pair programming * 40-hour week Collective ownership Metaphor	Customer on-site Planning game * Small releases	Testing Continuous integration Coding standards Refactoring * Simple design

The role of the RP perspective is discussed in depth only with respect to one practice in each group (marked with \*). That is, the focus is placed on pair programming, planning game, and refactoring. Specifically, in order to illustrate the potential contribution of a RP approach to software development that is guided by XP, we demonstrate a ladder of reflection for these three practices. We hope that, where possible, this illustration clarifies how the RP perspective may benefit the other practices in each category.

In the description that follows, it is assumed that readers are familiar with the XP practices [1].

#### 3.1. Team

It seems that the RP perspective fits well to parts of this group of practices. The rationale behind this assumption is that a reflective mode of thinking improves the comprehension of one's own thinking as well as of others' ways of thinking. As software development in general, and software development guided by XP in particular, are based on team interaction, it is reasonable to assume that the more one is

aware of mental processes and ways of thinking (of oneself or of the others), the more the teamwork is improved. In what follows the focus is placed on pair programming. The contribution of the RP perspective to the other XP practices that focus on the team is described briefly.

**Pair Programming:** This practice is one of the more discussed XP practices (cf. [15]). This practice should be applied firmly. In other words, “[a]ll production code is written with two programmers at one machine”. [1]. Benefits of this practice are presented in many research reports (cf. [9, 13]).

This practice specifies that any piece of code should be written by two developers, each of whom has a different role: the one with the keyboard and the mouse thinks about the best way to implement a specific task; the other partner thinks more strategically. As the two individuals in the pair think at different levels of abstraction, the same task is thought about at two different levels of abstraction *at the same time*.

In what follows we illustrate how a reflective mode of thinking may be introduced into the practice of pair programming. The illustration is based on the construction of a ladder of reflection during a pair programming session (see Table 1). The idea is to illustrate how a pair of programmers may increase the abstraction level of its thinking when reflection is interwoven within the process of software development.

**Table 1: A ladder of reflection: The case of pair programming**

Ladder rungs	Pair dialogue
Designing [ <i>a process of reflection-in-action</i> ]	A: Did we consider all the exceptions?
Description of designing [ <i>it takes the form of description with: appreciations, advice, criticism, etc.</i> ]	B: Good question. Let’s think about the best way to search for exceptions. I’m trying to understand what to think about when I’m looking for potential exceptions.
Reflection on description of designing [ <i>reflection on the meaning the other has constructed for a description he or she has given</i> ]	A: I think that this is not such a simple task. I have never thought about such systematic ways to look for exceptions. OK. Let’s give it some thought. [ <i>Working on formulating a systematic way for finding exceptions</i> ]
Reflection on reflection on description of designing [ <i>the parties to the dialogue reflect on the dialogue itself</i> ]	B: Now that we have developed a systematic way for finding exceptions, I think we must analyze these strategies and reflect on the path that led us to finding these guidelines.  A: Yes, this may improve our ability to solve problems of a similar nature in the future.

Looking at the various rows of Table 1, one may find that the subjects of reflection on each rung are objects of different levels of abstraction: While detailed elements are the focus on the first rung, ways of thinking are at the center of attention on the fourth rung.

**Sustainable Pace:** A reflective mode of thinking can be interwoven even in this simple-for-implementation XP practice. The working framework that this practice establishes enables one to detach oneself from the details involved in software development and, if one wishes, to reflect on what had happened during the day, without being swamped with details for long hours every day.

**Collective Ownership:** As the code is accessible to more minds, programmers must examine code that is written by others. Thus, they have to reflect and consider what reasons lead to specific decisions that their friends took while programming. In addition, when reviewing code that others wrote, developers may improve their understanding of their own code and its interface to the rest of the code.

**Metaphor:** The common use of metaphors is to bridge between a known domain and an unfamiliar domain. While thinking about an appropriate metaphor, developers must expand their perspective and analysis of the developed application. It is suggested that a RP perspective may improve developers’ performance in looking for an appropriate metaphor.

### 3.2. Customer

The literature is full of evidence of crises in software development processes. In many cases these crises result from some misunderstanding or other between clients and software developers. In other words, the client's needs and requirements are misunderstood, and as a result, the software system does not satisfy customer's needs. The following data illustrate this phenomenon: "Three quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer's requirements." [8]. Thus, addressing customers' ways of thinking is fundamental from the SE point of view. We suggest that a reflective mode of thinking like the one suggested by the RP perspective, may improve one's ability to understand the conceptions held by others in general and customer's needs in particular. The idea is that two processes occur simultaneously: A person improves his/her thinking about his/her mental processes; as the latter takes place, the person's understanding of his/her interaction with the environment is improved.

**On-Site Customer:** In the case of software development that is guided by XP, the customer is part of the development environment. As the customer is on-site for answering questions, it is suggested that when both customer and developers are guided by a reflective mode of thinking, developers, as well as customers, may improve their understanding of the developed application. This can happen, for example, when after some issue is clarified, the customer and the developers will reflect on their current understanding of the application vs. the one that preceded it.

**Planning Game:** One of the significant advantages of the planning game is that both the customer and the entire team participate in it, and thus all know the development process. Furthermore, guidelines that lead to decisions with respect to a specific release or iteration are clear to all. We take advantage of this fact and show how the fact that the customer and team define together the next release/iteration makes it possible to introduce a reflective mode of thinking. Table 2 presents a ladder of reflection which illustrates how this might happen.

**Table 2: A ladder of reflection: A planning game session**

Ladder rungs	A conversation during a planning game session
Designing [ <i>a process of reflection-in-action</i> ]	Customer: In fact, I want this feature to behave this way [ <i>moves her hands to illustrate</i> ]. Developer 1: Can you think about a similar feature you needed before?
Description of designing [ <i>it takes the form of description with: appreciations, advice, criticism, etc.</i> ]	Customer: What do you mean? Would you like me to think about a similar case in the past in which I wanted a similar feature? Interesting. I have never been asked to do something like this before. But yes, I can think about a situation in the past when we needed a new system for our inventory management. I wanted the application to have this feature and only when we received the system I realized that, in fact, what we need is something else, more ... [ <i>illustrates with her hands</i> ]. Let's call it B. Wow! Does that mean that we should not have at all the feature I described before?
Reflection on description of designing [ <i>reflection on the meaning the other has constructed for a description he or she has given</i> ]	Developer 2: We do not know. We can check the two options. But, can you please recall, what, in the case you just mentioned, led you at the end to realize that what you need is B, and why you didn't (or couldn't) realize this before, I mean, before you got the system and started working with it. Customer: Truly, the problem was that we did not consider the full setting in which the system would work. I think that we should consider the same issue now, before I make the final decision.  [ <i>The customer and the developers think about the way the application will be used, focusing on the specific considerations that were neglected in the customer's previous experience. At the end they decided about a third option that should be applied for these specific circumstances.</i> ]

Reflection on reflection on description of designing <i>[the parties to the dialogue reflect on the dialogue itself]</i>	<p>Customer: It's amazing. I must trace with you the full path we went through together.</p> <p><i>[The customer and developers dedicate the next 15 minutes for this purpose].</i></p> <p>Customer: I do not want even to think about the catastrophe that could have happened if you develop one of the first two options we talked about. I must learn the lesson. First of all, I'd like to apologize for my resistance to take part in the planning game. I must confess that only now I understand how I should manage the all business with the new application.</p> <p>Tracker: I think we also learnt something from this experience. First, we should not be afraid to ask our customers difficult questions and to insist on getting answers. Second, the specific circumstances you introduced us to may be useful in our future projects. Finally, we should remember that before making final decisions and moving on, sometimes it is worth checking whether we consider all options. I believe that eventually, even if we stay with the first option, this would not be considered a waste of time.</p>
--	--

As can be observed, the customer improves her understanding throughout the planning game scenario described in Table 2. In fact, she got her insight only when she was asked to reflect about similar situations in the past in which she needed similar features. It is not argued that hadn't she asked to reflect on past experiences she would not have recalled such cases. However, it is plausible to assume that this insight would have arrived later (maybe only after an inappropriate feature would have been developed). As can be observed, the team also improved its communal understanding with respect to decision-making processes and guidance of customers in describing their needs. The contribution of such lessons to software development processes is clear.

**Small Releases:** Beck [1] tells us to “[p]ut a simple system into production quickly, then release new versions on a very short cycle.” Such an experiment invites a reflective mode of thinking very naturally. In fact, the small releases are introduced to let developers re-examine their progress in small cycles. We suggest that the use of a RP perspective on those small releases may even strengthen the risk management that results from keeping the releases small. Specifically, we suggest that the question to be answered usually - “Does this release carry us toward the eventual goals of the project?” - can be answered more easily by a RP approach.

### 3.3. Code

Based on arguments that address architectural creations, it is suggested that the reflective perspective may improve the application of those XP practices that concern with the code: testing, refactoring, simple design, continuous integration and coding standards. In this argument the code is viewed as the analogical object to the architectural creation, for which the RP perspective has been developed originally. Thus, as a RP perspective may improve the creation of the architectural creations, it is suggested that the application of a RP perspective in those XP practices that deal with the code, may improve the code. Specifically, if a reflective mode of thinking is interwoven into each of these practices, the team may improve the code it produces and the result would be a code that is more correct, more readable and easier for future maintenance.

**Refactoring:** In our opinion, as this practice is so similar to process of re-design in the case of architectural creation, it may be largely benefited from the RP perspective. Table 3 illustrates a ladder of reflection in the case of refactoring. The dialogue here is conducted during a refactoring session in which a pair of programmers changes a procedural design to an object oriented design (cf. [3]).

We expand the discussion of the application of a RP in refactoring by quoting Kent (in [3]): “[Refactoring] is like a new kind of relationship with your program. When you really understand refactoring, the design of the system is as fluid and plastic and moldable to you as the individual characters in a source code file. You can feel the whole design at once. You can see how it might flex and

change – a little this way and this is possible, a little that way and that is possible.” (p. 410). This quote is chosen as its spirit is similar to the way artists talk about their creation process. Possible conclusion would be that if refactoring is similar to one aspect of artistic creation processes, and if the later find the RP approach helpful, software developers may introduce a RP perspective into development processes in general and into refactoring sessions in particular. Needless to say that many computer scientists view software development like art. From reasons of space limitation we quote here only Knuth who says, with respect to the artistic nature of programming, that “[t]he 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. ([7], p. V).

**Table 3: A ladder of reflection: A refactoring session**

Ladder rungs	A dialogue during a refactoring session
Designing <i>[a process of reflection-in-action]</i>	<p>Developers 1: We were told that the system was originally developed by a procedural approach. Right?</p> <p>Developer 2: Yes.</p> <p>Developer 1: And we should change it to OO.</p> <p>Developer 2: Ya.</p> <p>Developer 1: I heard about Fowler’s book that explains very precisely how to do it.</p> <p><i>[Worked according to Fowler’s book ([3]) and changed the design. All the tests passed.]</i></p>
Description of designing <i>[it takes the form of description with: appreciations, advice, criticism, etc.]</i>	<p>Developer 2: Let’s think what we have done.</p> <p>Developer 1: Why? We already did it and all the tests passed.</p> <p>Developer 2: Yes, but we should get something out of all this work, beyond the code. Some wisdom that will help us in future development. I’m sure that there is some wisdom in such processes that is beyond the actual change of the code.</p> <p>Developer 1: OK. Let’s try to identify the moment when each of us realized what we are doing; I mean, when we really understand what the purpose of all this change is.</p>
Reflection on description of designing <i>[reflection on the meaning the other has constructed for a description he or she has given]</i>	<p>Developer 1: If I have to indicate the moment I understood why all this work is worth something, I would say that it was the moment I realized that there are methods different than <code>set</code> or <code>get</code> methods. When we started typing <code>public Boolean isFull</code> I got it.</p> <p>Developer 2: So, can we decide that in the future, before we start working on a task of this kind, we will check first whether there are methods different than <code>set</code> and <code>get</code>? You know what, let’s try it for what Joe just asked me to check.</p> <p><i>[Checked and realized that in this case it would not be worth to dedicate the 5 hours needed for such a change. Moved to their next task].</i></p>
Reflection on reflection on description of designing <i>[the parties to the dialogue reflect on the dialogue itself]</i>	<p>Developer 1: Let’s adopt it as our habit-of-mind when such tasks come up. I think that if we had known this lesson before we started all this stuff, we could improve our work and make it more productive, not to mention all the hours that we could save.</p>

**Testing:** Beck defines the practice of testing as follows: “Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.” We add and say: “Reflect on how you test and on what you learn from it and improve your understanding of testing processes”.

**Simple Design:** Rasmuson ([9]) says that “[i]f simplicity is the destination, refactoring is the vehicle for getting us there.” Based on the above illustration, it is clear how a RP approach may support this practice. In fact, the source of RP is in examining design processes. Specifically, a reflective mode of thinking may improve developers’ understanding of what simple design consists of.

**Continuous Integration:** If one reflects of the integration process, his or her understanding of the code may be improved. Again, similarly to the case of Small Releases, a reflection on the continuing goals of the project helps determine how close integration is to the completion of the software.

**Coding Standards:** The RP perspective in this case may be expressed by the way a team chooses/develops its coding standard. After all, according to the XP’s attitude of “they’re just rules”, “[t]hey are the rules that the team embraces” (look at Jeffries’s essays “*They’re just rules!*”, <http://www.xprogramming.com/Practices/justrule.htm>). The coding standards can be formulated based on a reflective process in which the team establishes the standards that fit its own communication style. After these conventions are set, a reflection may be helpful in cases where a particular convention seems not to achieve its targets: a reflective mode of thinking may help in understanding the source of this mismatch.

## 4 Conclusion

In this article we suggest to add the practice of reflection to XP practices. It is argued that a reflective mode of thinking may improve software developers’ understanding of their own (and their teammates) ways of thinking and, as a result, they may improve both the way they develop software and their understanding of the development environment. In our opinion, a reflective mode of thinking is especially suited to the community of XP since XP encourages collective knowledge which, in one way or another, forces software developers to understand the other person’s ways of thinking. Such a reflective mode of thinking may also solve such problems as the following, suggested by Glass ([4]) in the context of the industry/academia communication chasm: “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). It is suggested that practitioners’ reflection on the way they solve problems (when conducted on a regular basis) may help in real-life situations.

The focus in this paper is placed of constructing ladders of reflection and on how they may improve software development processes. Our suggestion is to explore specific ways (or maybe even, procedure) for making reflection an integral part of XP both in the industry and in the academia. In the academia a reflective practice perspective can be integrated naturally into software engineering programs which include students’ software projects that are developed by XP. It is suggested to add activities which induce students to reflect on the way they develop software systems. Within the scope of students’ projects, such tasks can be offered to students in different activities such as design, coding, and testing.

## REFERENCES

1. Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.
2. Cockburn, A. (2001). *Agile Software Development*, Addison-Wesley.
3. Fowler, M., Beck, B. (Contributor), Brant, J. (Contributor), Opdyke, W. and Roberts, D. (2002). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.
4. Glass, R. L. (1997). Revisiting the industry/academe communication chasm, *Communication of the ACM* **40**(7), pp. 11-13.
5. Hazzan, O. (2002). The reflective practitioner perspective in software engineering education, *The Journal of Systems and Software*, **63**(3), pp. 161-171.

6. Kerth, N. L. (2001). *Project Retrospectives: A Handbook for Team Reviews*, Dorset House.
7. Knuth, D. E. (1969, 2nd Printing). *The Art of Computer Programming*. Addison-Wesley.
8. Mullet, D. (July, 1999). The Software Crisis, *Benchmarks Online - a monthly publication of Academic Computing Services*, a division of the University of North Texas Computing Center **2**(7), <http://www.unt.edu/benchmarks/archives/1999/july99/crisis.htm>.
9. Rasmusson, J. (2002). Strategies for introducing XP to new client sites, *Proceedings of the XP/Agile Universe 2002*, LNCS 2418, pp. 45-51.
10. Schön, D. A. interviewed by John Bennet (1996). Reflective conversation with materials. Terry Winograd, *Bringing Design to Software*, Addison-Wesley, pp. 171-184.
11. Schön, D. A. (1983). *The Reflective Practitioner*, BasicBooks,
12. Schön, D. (1987). *Educating the Reflective Practitioner: Towards a New Design for Teaching and Learning in The Profession*, San Francisco: Jossey-Bass.
13. Shukla, A. and Williams, L. (2002). Adapting Extreme Programming for a core software engineering course, *Proceedings of Conference of Software Engineering Education and Training - CSEE&T 2002*, pp. 184-191.
14. Singer, A. (1994). Towards a definition of software design, *Design+Software - The ASD Newsletter*, <http://www-pcd.stanford.edu/asd/info/articles/definition.html>
15. Williams, L. A., Kessler, R. R. (2000). All I really need to know about pair programming I learned in the kindergarten. *Communications of the ACM* **43** (5), pp. 108-114.