Weaving Experiences from Software Engineering Training in Industry into Mass University Education

Wolf-Gideon Bleek, Carola Lilienthal, Axel Schmolitzky
Department of Informatics, University of Hamburg
Hamburg, 22527, Germany
{bleek,lilienthal,schmolitzky@informatik.uni-hamburg.de}

Abstract

Basic software engineering education is an important part of IS education. This paper shows and critically discusses how experiences gained from years of software engineering training in the industry can be transferred to mass university education. The approach relies on cyclic, iterative, and problem based learning and puts equal stress on technical skills (such as object-oriented and database programming) and on soft skills (such as presentation techniques, handling personal conflicts and cooperating in a team).

Keywords: software engineering, programming education, concepts for teaching

1. INTRODUCTION

Basic software engineering education is an important part of IS education. At the core of any software engineering education stands a solid foundation in programming, nowadays with a strong focus on object-oriented programming. But a well-educated software engineer should not only be a competent software developer, she should also possess several soft-skills: presenting and discussing software designs and architectures, cooperating in a team, managing personal conflicts etc.

The Software Engineering Group at the University of Hamburg has gained experience in training people in object-oriented programming and software engineering concepts in both the industry and at the university for quite some time. The group has been responsible for educating undergraduate students in basics of imperative and object-oriented programming since 1994. In addition, the same people have been training software engineers in the industry for more than ten years.

While both types of training cover the same topic, the way a training is performed is extremely different. At university, students are instructed by a combination of lecture (approx. 250-300 people) and tutorials (approx. 20-25 people each with no hands-on programming, just programming homework). In the industry, our trainees are educated by an interwoven mixture of short lectures, small-group exercises and practical programming work.

Our impression from comparing the effectiveness of both approaches is that company trainings work much better, which is not surprising. In consequence, we transfer as much of our industry training concept, validated by experience, to university education. We are constantly trying to adapt our best practices to this different environment, while being aware that resources are sparse at university.

In this paper, we will first present our experiences gained from years of training in the industry. We will then extract the concepts and show how they have been transferred to the organizational structure of a mass university with undergraduate and graduate education. After that we outline the evaluation results of two surveys performed in the years the new concept was completely installed. In the discussion we critically review the experiences, differences, and problems observed. These lead to improvements that will further enhance our teaching concepts.

2. THEORETICAL BACKGROUND

The Software Engineering Group at Hamburg University is responsible for training students at the undergraduate and graduate level. Its focus is on interactive application software by adopting a human-centered approach, emphasizing cooperation with users, evolutionary development strategies, and object-oriented software construction.

Evolutionary and cyclic development methods [14] have proven to be suited best for developing software for socio-technical systems, i.e. interactive software. These methods foster communication between all relevant participants and promote a mutual learning process.

Communication and active learning play an important role in educating software engineers (see below).

On the side of learning we follow Piaget's idea ([16], [17]) of acting towards a goal, collecting experiences, and critically reflecting these experiences to build up knowledge that can be used as the basis for new learning. This is a cyclic approach as well. The learning environment that is provided should support these activities by providing a realistic setting on the one hand and on the other by giving space, freedom, and skilled support at need.

To constantly reflect on and improve our work of teaching people in software development, we concurrently reflect on the teaching experience. This is methodologically founded on Action Research [2], [15]. A cycle of action planning, taking, evaluating, specifying the learning, and diagnosing is followed. We support our observations by regular surveys and offer people to engage in the design of the teaching situation.

3. EXPERIENCES FROM IN-HOUSE TRAINING IN THE INDUSTRY

Before we describe our experiences with software engineering training in the industry, a few words on the background are necessary.

The company responsible for the industry training, *it-wps GmbH*, is a spin-off company at the Department for Informatics. Several staff members of the Software Engineering Group work part-time at it-wps as well, one professor is the CEO. The company has its main profile in object-oriented software construction, consulting and training.

In the following we describe the didactical aims of our industry trainings, our experience with time constraints, the structure that evolved over the years and finally sum up our core principles and best practices.

Didactical Aims
Professional software development implies team work. The technical skills of the project team members are important, and typically lots of money are spent to increase the technical knowledge of software development teams. But several soft skills are at least equally important for the success of a project: software engineers need to be flexible and communicative, they must be used to giving presentations, they need to keep the whole picture in mind and take responsibility for the project they are part of. These and other soft skills are much too often forgotten when educating software engineers.

We therefore put equal emphasis on three aims when training software engineers:

- Sound technical knowledge (e.g. programming languages, design patterns, algorithms, databases, web-technologies).
- Sound methodological knowledge (e.g. development of a quality architecture and design, software development process models).
- Improved soft skills (e.g. presentation techniques, giving and receiving critique, a culture of continuous feedback as part of a development process, handling of personal and group conflicts, taking responsibility, leading and coordinating teams, learning to learn).

Constraints
In training professionals we had various arrangements with different companies to achieve these didactical aims. The two main variables with all trainings were:

- Time available for the training program, and
- Skills the trainees already brought along.

If the trainees are novices in object-oriented programming, a training program shorter than nine weeks will surely not at all help them to reach the three didactical aims. From our experience, novices need at least a seven-month training program to really reach an appropriate level. Trainees who already know object-orientation have been trained in about five weeks. In this article, we try to transfer our experience from company trainings to undergraduate student education. Therefore we will from now on refer to our experience with the training of novices.

Structure
Over the years our industry training of novices evolved more and more to comprise of three parts. If possible under the given time constraints, these parts are:

a) *Teaching in lab* (three and a half months), intermixed with a high percentage of exercises.

b) A *mini project* (two weeks), to experience a complete software development process.

c) A *real in-house project* (three months), performed by the trainees as a project team.

The teaching-in-the-lab part of the program starts with an intense classroom setting. Most of our classroom weeks follow a 3-2-day-pattern: for 3 days the trainees are provided with the relevant information. The trainer presents slides and assigns small exercises ("lecture days"). After 3 days of "information input" and small feedback cycles, the content is deepened for 2 days with larger assignments ("deepening days"). While the trainers change, depending on the topics taught, one dedicated trainer serves as a permanent tutor, monitoring the deepening days during the whole curriculum.

The mini project introduces team work and imparts a first impression of a full project. Here we use XP [3] as the development methodology, as it puts a clear focus on programming, but has *communication* and *feedback* as two of its core values at the same time. The trainer has to play the part of the customer and to act as a coach.

To prepare the real-in-house-project part of the program, we examine all open projects in the company and choose one that is relevant but not critical. By working in an in-house project with real customers the trainees are able to become project team members instead of just programmers. They experience typical project problems on different levels: social, technical, organizational etc. Even more than in the 4-month-curriculum, the soft skills become highly important. The project allows us to conduct several best practices and following our core principles.

Core Principles and Best Practices

To achieve our didactical aims, we apply several principles and practices. Some of them are more important during teaching in the lab, some are more relevant during the in-house project. The trainers, nevertheless, have to be aware of them during the whole program.

1. Concepts over API details.
   In the first part of the program the main technical knowledge is imparted. As technology changes with high speed, we focus on conceptual knowledge rather than on exact knowledge of every technical API. This way a technical basis can be created and the trainees are enabled to learn technical API details by themselves. Especially during the in-house project, where trainees are often faced with various backend systems, this approach normally turns out to be most valuable.

2. Objects first – reality next.
   We always use the learning environment BlueJ to introduce the concepts of object orientation [1]. Later on the actual development environment of the company is introduced. One of the major goals in teaching beginners is to get them in touch with the fairly intuitive idea of objects. BlueJ permits starting their education with object-oriented topics and delaying programming language-dependent problems. All our training programs benefited from BlueJ [8].

3. Permanent availability.
   During the first part of the program one or two trainers are attending the trainees at all times. If the number of trainees exceeds 10 (programming in 5 pairs), two trainers are mandatory. During the in-house project, the trainees start to work on their own. At least once a day a trainer observes the results. The role of the trainers shifts now to that of a consultant.

4. Learning by doing.
   We use as few slides and as much exercises and practical work as possible. This includes exercise periods of one to several days, the software projects, moderation of discussions, and presentation of work results [10]. The role of the teacher is - according to constructivist learning theory [11] - more that of a link man helping to foster a learning process. When necessary, the trainer helps out as a specialist [9] introducing a technology or concept. In our experience, the learning curve increases this way, and any problems in following the curriculum are encountered early.

5. Iterative and incremental learning.
   The typical tasks of a software engineer are practiced right from the beginning with small exercises. The tasks are then deliberately repeated all through the program with increased scope and complexity. In our experience, to approach a topic cyclically, profoundly deepens the understanding and intensifies the memory.

6. Permanent reflection.
   Training periods introducing new concepts are followed by moderated plenum discussions and summarizations. The trainer is enabled to monitor the progress and to adapt the curriculum if needed. The trainees maintain written notes on the terms learned. The trainees improve their ability to discuss the subjects learned.

7. Intense and personal feedback.
   Continuous feedback is given on several levels by trainers and trainees. This increases the learning curve, as problems are discovered earlier. If necessary, special promotion of weaker trainees is applied.

8. Soft skills addressed explicitly.
   The trainees' soft skills are addressed directly. If trainees present results, trainers give feedback not only on the actual work results, but also on the presentation style. In our experience, this approach turns out to be very successful, particularly during the in-house project. Normally problems with soft skills become apparent in this phase of the training, e.g. some trainees are unable to take on responsibility or find it difficult to compromise. If the trainers find a way to address these problems respectfully, the soft skills of the trainees improve noticeably.

9. Learning environment close to the future job setting.
   In company training, we try to choose a learning environment that is as similar as possible to the trainees' future workplaces. However, we start with classroom teaching and small exercises. The similarity to the future workplaces is restricted to the task of programming. In the in-house project, the trainees additionally interact with in-house customers, plan project iterations and develop a software architecture designed according to the company's technical strategies.

10. Pair programming
    All programming in the lab and in the projects is done in pairs. Pairs have to change at least once a day. This XP practice [3] is related with principles 5, 6, 7 and 8, because it forces the trainees to work together and talk about what they are doing.

All core principles and best practices have helped our trainers to reach the aims of the training programs we have performed in the industry.

4. TRANSFERRING THE CONCEPT

With the good results experienced by applying our industry training concepts and the growing number of drawbacks that we observed at the university, we initiated transferring the concept to the university.

Obviously, the university is a totally different environment. However, even university teaching varies between countries, so we will first sketch the general conditions at a German university. We will then adapt all the elements of the in-house concept to the other environment by outlining their arrangement. After that, we take a look at their integration and rounding off of the concept.

German University System – A Classical Teaching Structure
At German universities, students have to spend a number of weekly "university hours" (45 minutes) for each course. A one semester course typically spans 14 weeks. From a student's perspective, a lecture with 2 hours represents 90 minutes of attendance and requires about the same amount of time for preparation. In contrast, an adjunct tutorial with 2 hours represents 90 minutes of attendance and 180 minutes of preparation.

A typical set-up is a 2-hour lecture combined with a 2-hour tutorial. Students hear about a topic in the lecture, go to the tutorial, get an exercise sheet on that topic and talk about questions related to it. During the week, students spend their time on solving the exercises, either at home or in unattended labs. In the next week's session, they present their solution in the tutorial and tutors collect their work. The new exercise sheet is dispersed. After another week, the students get their exercises back with comments from the tutor. Common problems are discussed. Students are typically encouraged to work in groups of two to four on each sheet.

Studying Informatics in Hamburg
Hamburg University is a public university, open to anybody with a high school degree. The undergraduate curriculum in Informatics starts every year in fall. The number of beginners ranges from 220 to 400 each year, depending on several external factors. With regard to education in programming concepts, students take three courses – P1, P2, and P3 – that currently cover the range from logical and functional programming (P1), imperative and object-oriented programming (P2), and advanced programming concepts, such as concurrent programming, database programming, and transactions (P3). The structure of each of these courses is a general lecture combined with tutorials.

From the teaching perspective, each combination of lecture and tutorials for approx. 300 students requires staff of about 15 people. In addition to a professor, who is responsible for the content of the whole course and giving the lecture, another person is in charge of organizing all related formal and conceptual activities (e.g. managing the registration, writing the certificates), and about thirteen staff members and graduate students are engaged as tutors in the tutorials. From the students' perspective, the combination of lecture and tutorial follows the general pattern sketched above.

Drawbacks of the Classical Approach
The classical teaching structure does not work well for programming courses. One evidence is the course P2, conducted by the Software Engineering Group.

In conducting this course, we faced the following problems:

- The old concept required repeated efforts in designing exercise sheets (to counteract the students' activities in providing answer-sheets for the following generations). This increased the workload of the tutors, as they were traditionally integrated in this design process.
- Correcting student's work poses a significant workload of approx. 6 hours per week for each tutor.
- There is a minimal relation to real-life projects in the exercises.
- Exercises take no or little account of previous knowledge.
- Most of the time, the tutorial time is too short.
- There is too little individual feedback, usually just in written form on paper or per email.
- Students' working groups tended to favor division of labor instead of group work to get their exercises done.
- There is too much room for cheating.

In summary, the outcome of our university course did by far not rectify the effort we put into preparing and conducting it. Moreover, many students did not demonstrate sufficient understanding of the core concepts in oral exams. In general, the classical approach suffers due to the following facts:

- Learning targets that go beyond technical questions are hard to teach in this kind of setting.
- The interrelation between analysis, design, and construction in software development is hardly teachable in exercises designed for weekly sessions.
- Object-oriented programming requires a lot of conceptual understanding and practical experience.
- Building larger object-oriented systems implies teamwork.
- The period of three weeks for presenting the exercises, collecting students' work, and handing out

corrections if far too long to evoke a learning process.

Mapping from Industry Training to University Education

In the setting described above, it is apparent that we should make use of our experience in industry training. The question then is: How well can we map the principles and practices, working with the given resources (personnel, training rooms etc.).

Bearing in mind the time necessary to train novices and to meet the special conditions of the university, we have mapped our industry training to a number of courses spanning from undergraduate to graduate studies: one undergraduate lecture with attended labs (P2), an undergraduate mini-project, a graduate project, and an optional industry internship.

The emphasis in these courses changes according to the students' maturity and the settings' appropriateness.

With several years of IT experience, we have learned that programming languages and APIs come and go. It is therefore imperative to concentrate on the concepts of programming languages and of basic APIs and apply them by using an exemplary language. We favor Java as our teaching language. Java is consistently used in all courses conducted by the Software Engineering Group.

Lecture with attended labs. The aim of this course is to lay a well-founded understanding of imperative and object-oriented programming. It is therefore crucial that students get a consistent and deepened picture of this topic as their first impression. In classic tutorials, students usually pay little attention and do not have the necessary degree of involvement. To overcome these drawbacks, we have implemented an undergraduate course (P2) with "intense attendance" (3) by providing weekly tutorials of 180 minutes, in which students do actively program in attended labs. This fosters "learning by doing" (4), as students have to finish their exercises within the given time. In this setting, we are able to give "intense and personal feedback" (7) that allows us to monitor students' progress as well as problems they face.

By asking students to work in pairs, we provoke the advancement of their inter-personal skills ("pair programming" (10)). They are urged to ask questions and answer them, they have to legitimate their actions, and they gain respect for each other's actions. This encourages "permanent reflection" (6) on the topic.

We follow the rule "concepts over API details" (1) in that we not only waive exercises that ask for specifics of an API, but also foster students to acquire skills on searching for the information with, for example, the API documentation.

The first eight weeks (phase one of the course) are dedicated to basic programming concepts. Objects First [1] is applied as the teaching method. The last six weeks (phase two) provide a continuous theme covered by a more complex development activity. Here, problems in programming that are related to complexity become noticeable for the students.

To meet the principle "best tools for the task" (2) we have the liberty to choose from a variety of tools available at the university. The BlueJ [8] environment was therefore our first choice to start teaching an object-oriented programming language. In the second phase we introduce Eclipse [13] as a development environment to confront students with a professional tool. To be able to freely choose the development environment is an advantage compared to industry trainings, as we usually have to use the company-wide tool without assessing its suitability.

Mini-Project. In the undergraduate mini-project, conducted in the three-months semester break between semester 2 and 3, we intensify programming work by letting the students work on one single project for 5 full days in one week. Teamwork is required because the task is too large for individuals or pairs. 3 or 4 project teams work on the same task independently, each consisting of 10 to 14 members. Most XP practices are used in this project and the students work with a repository (via the CVS integration in Eclipse) for the first time. This addresses most effects related to group dynamics. In addition to that, we consider "soft skills as an explicit topic" (8) by letting the teams give presentations of their final results in front of the whole course.

The overall concept of undergraduate teaching stresses "iterative and incremental learning" (5). The exercises gradually build upon each other; describing one concept requires understanding a lower lever concept (e.g. reference and polymorphism). Moreover, students are urged to secure the necessary information on demand.

Some didactical means obviously cannot be met in the undergraduate courses, e.g. the "learning environment close to the future job setting" (9). To put this means into practice, we offer projects and internships at the graduate level with our industry partners.

Graduate project. The focus of the graduate software engineering project is to provide a complex software development activity, which spans two semesters. We have discussed the concept and some of our experiences in [7].

Industry internship. The industry internship is a voluntary activity that offers committed students the possibility to face real-life software developing conditions. The specifics of that course cannot be covered in this paper.

5. EVALUATION OF THE FIRST TWO YEARS

Graduate projects based on object-orientation are a core part of our graduate education since 1996. We have been conducting the undergraduate mini-project following XP practices since 2000. We applied the new concept for P2

for the first time in 2003 and for the second time in 2004. So, our experience with projects is well established and reflected ([7] and [6]), whereas the P2 concept with attended labs is a fairly new experience ([4] and [5]).

During both semester breaks after the new P2 course, we conducted and evaluated three surveys: one for students who had finished the course successfully (pass or better), one for students who had not passed and one for the instructors.

Students' Feedback
Each year, we conducted two anonymous surveys where the students could tick pre-formulated statements and give individual feedback as well.

The feedback of the students who passed was very positive. 92% did NOT tick the statement "*I would have preferred a traditional course with tutorials and homework*". 82% ticked the statement "*The lab classes were a lot of fun*". Only 38% ticked "*The lab classes were hard work*". The students had a very good impression of the instructors: 93% ticked the statement "*My instructors were well qualified for their job*". In the free-text answers most students claimed that there was not enough time for individual feedback and that the instructor/student ratio should be higher.

The feedback of the students who have not passed showed that the new concept had little impact on failure. Only 16% (3 out of 19) blamed it directly for their failure, while most claimed personal problems or external pressure. One even had passed the year before, was just curious about the new concept and claimed that he liked it better than the old concept.

Instructors' Feedback
We asked several questions in an anonymous survey both years. We were especially interested in the instructors' workload due to the new concept. The majority (6 out of 11) replied that the new concept consumed less or the same amount of time than the old concept, 3 abstained. Most (8 out of 11) said that their personal stress level during contact hours was ok or even less than before, 3 found the new concept more strenuous.

In the first year, most instructors criticized the instructor/student ratio as too low. Quite often time ran out towards the end of a lab session, when many students wanted their work to be assessed.

7 out of 11 instructors saw the new concept as a substantial improvement, 4 abstained, none saw it as a drawback.

General Observations
Most of the (few) problems of the first year could be avoided in the second year:

- The exercise sheets just had to be polished in the second year, which relieved the instructors substantially.
- The faculty granted more instructors, leading to an improved instructor/student ratio.

One of our observations with the old concept, that the interrelation between analysis, design, and construction in software development is hardly teachable in exercises designed for weekly sessions, still applies to the new concept. Students get in contact with this issue for the first time in the mini project, but at that time already within a group of collaborators. There is something missing in between, where individuals experience how a concrete, real-life problem can be solved by software that is designed and implemented after an analysis of the problem.

We observed that instructors from outside the Software Engineering Group that had no experience with industry training did not cope well with the intensity of the lab work. Presence in two 3-hour lab sessions is stressful for most of them; four 3-hour lab sessions are almost borderline. In contrast, staff members that are used to full-day teaching in the industry were less affected by this.

Instructors in an attended lab obviously need to have comprehensive programming experience, in our setting especially with imperative and object-oriented programming. As experience within these paradigms can still not be taken for granted, instructors have to be selected carefully. Some graduate student instructors are better suited for the job than some of the university staff members that grew up programming in a different paradigm.

6. DISCUSSION

Overall, we claim that so far our experience in industry training could be well transferred to university education. But there are still some points that need to be addressed:

- Soft skills in presenting small designs.
- Sometimes high discrepancy in skills of pair members.
- Instructors' skills are not necessarily specialized on the topic taught.
- Organizational drawbacks (tutors and students have problems in attending courses with atypical timeframes, holdup on reviewing exercises at the end of a session).

The focus of the P2 course is on core programming practices; almost no soft skills (beside feedback within programming pairs) are addressed. In the old concept, students had to present their solutions in the tutorials in front of around 20 people. We sacrificed this for the sake of improved programming skills, but we are aware that in future settings oral presentations should be on the agenda again.

In the near future the Informatics faculty in Hamburg has to switch to a new structure that is much closer to a Bachelor education in the Anglo-Saxon parts of the world. The current plan includes the abolishment of functional and logic programming in the first year and its replacement with more imperative and object-oriented basics. We appreciate this shift in focus, as it allows us to elaborate on some important points that so far could not be covered due to time constraints.

7. CONCLUSION

In this paper we have tried to present our experience with transferring successful training in the industry to university education. For the first time we did this not just for one course, but tried to present the big picture, ranging from first-year programming to graduate projects and industry internships. We think that in any computer-related education, may it be computer science, software engineering or information systems, object-oriented analysis, design and programming together with the soft skills to present and explain the resulting software artifacts, should play a major role. We think that industry training can be the major source for innovation in university education, if the staff can gain experience in both worlds.

8. ACKNOWLEDGMENTS

We thank Heinz Züllighoven for the freedom and support he gave us in designing the courses. Additional thanks go to Petra Becker-Pechau for her active participation and work to implement these courses.

9. REFERENCES

[1] Barnes, D.J., M. Kölling: Objects First with Java – A Practical Introduction using BlueJ, 2nd Ed., Prentice Hall / Pearson Education, New York, 2004.

[2] Baskerville, R. L. Investigating Information Systems with Action Research. Communications of the Association for Information Systems. Volume 2, 19, October 1999

[3] Beck, K. eXtreme Programming - Embrace Change. Addison-Wesley, Boston, MA, 2000.

[4] Becker-Pechau, P.; Bleek, W.-G.; Lilienthal, C.; Schmolitzky, A., "Educating Non-Programmers to Flexible, Communicative Software Engineers in a 10 Month Training Program", 17th Conference on Software Engineering Education and Training (CSEE&T 2004), Norfolk, Virginia, 2004

[5] Becker-Pechau, P.; Bleek, W.-G.; Schmolitzky, A.; Züllighoven, H., „Integration agiler Prozesse in die Softwaretechnik-Ausbildung im Informatik-Grundstudium", Software Engineering im Unterricht der Hochschulen (SEUH) 2003, Berlin; In: dpunkt-Verlag, 2003

[6] Becker-Pechau, P.; H. Breitling; M. Lippert; A. Schmolitzky, "Teaching Team Work: An Extreme Week for First-Year Programmers", In: Michele Marchesi, Giancarlo Succi (Eds.), Extreme Programming and Agile Processes in Software Engineering, Proceedings of 4th International Conference XP 2003, Genova, Italy, May 2003, Springer, LNCS 2675, pp. 386-393., 2003.

[7] Bleek, W.-G., G. Gryczan, C. Lilienthal, M. Lippert, S. Roock, H. Wolf, H. Züllighoven „Von anwendungsorientierter Softwareentwicklung zu anwendungsorientierten Lehrveranstaltungen - der Werkzeug & Material-Ansatz in der Lehre", Software Engineering im Unterricht der Hochschulen (SEUH) 99, Berichte 52, B. Dreher/Ch. Schulz/D. Weber-Wulff (Hrsg.), Workshop des German Chapter of the ACM und der Gesellschaft für Informatik (GI), pp. 9-20, 1999

[8] BlueJ. www.bluej.org (last visited June 30 2004)

[9] Brown, A. L., Ash, D., Rutherford, M., Nakagawa, K., Gordon, A., Campione., J. C. Distributed Expertise in the Classroom. In Salomon, G. (publ.): Distributed cognitions: psychological and educational considerations. Cambridge University Press, UK, 1993, pp. 188–228.

[10] Clark, H. H., Brennan, S. E., Grounding in Communication. In Resnick, L., Levine, J. M., Teasley, S. D. (eds.): Perspectives on Socially Shared Cognition. Washington, USA, 1991.

[11] Cockburn, A., Agile Software Development, Addison-Wesley, Boston, 2002.

[12] Dewey, J. (1933) How We Think: A Restatement of the Relation of Reflective Thinking to the Educative Process. Chicago, USA.

[13] Eclipse. www.eclipse.org (last visited June 30 2004)

[14] Floyd, C., F.-M. Reisin, and G. Schmidt. Steps to software development with users. In C. Ghezzi and J.A. McDermid, editors, ESEC89, number 387 in Lecture Notes in Computer Science, pages 48–64. Springer-Verlag, Berlin, 1989.

[15] Johnson, B. Teacher-As-Researcher. ERIC Digest. ERIC Clearinghouse on Teacher Education Washington DC. ED355205. 1993

[16] Piaget's theory. In P. Mussen (ed) Handbook of child psychology, Vol.1. New York: Wiley, 1983.

[17] Studies in reflecting abstraction. Hove: Psychology Press, 2000.