

Transferring Experience from Software Engineering Training in Industry to Mass University Education – The Big Picture

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

Abstract

This paper shows and critically discusses how experience gained from years of software engineering training in industry can be transferred to mass university education. The approach relies on cyclical, iterative and problem-based learning and places equal emphasis on technical skills (such as object-oriented and database programming) and soft skills (such as presentation techniques, handling personal conflicts and cooperating in a team context).

1. Introduction

An essential element of all software engineering education is a sound grounding in programming, with a strong focus today on object-oriented (OO) programming. But well-educated software engineers should not only be competent software developers, they should also have a number of soft skills: the ability to present and discuss software designs and architectures, to cooperate in a team context, to manage personal conflicts, etc.

The University of Hamburg's Software Engineering Group has gained experience in training people in OO programming and software engineering concepts in both industrial and academic settings over a considerable period of time. Since 1994, the group has been responsible for teaching undergraduate students the fundamentals of imperative and OO programming. The same people have been training software engineers in industry for more than ten years.

While both types of training cover the same subject-matter, there are significant differences in the way the training is conducted. At universities, instruction is through a combination of lectures (with approx. 250-300 students) and tutorials (each with approx. 20-25 students; no hands-on programming, just programming assignments). In industry, our trainees are taught through an interleaved system of short lectures, small-group exercises and practical programming work. The impression we have gained from comparing the effectiveness of the two approaches is that industrial training works much better. This is not surprising. As a result, we are transferring many elements of our industrial training concept, which is validated by experience, to university education. We are constantly trying to adapt our best practices to the academic environment, bearing in mind that resources there are limited.

In this paper, we begin by outlining our teaching background and our experience acquired by training people over many years in an industrial context. We then define the basic elements of our concept and show how they have been transferred to the organizational structure of a mass university providing undergraduate and graduate education. This is followed by an analysis of the results of two surveys conducted in the years during which the new concept was properly implemented. We conclude by critically discussing the experience gained, the differences encountered and the problems observed. This will help us to improve and further enhance our teaching concepts.

2. Teaching Background

The University of Hamburg's Software Engineering Group is responsible for training students at the undergraduate and graduate level. It focuses on interactive application software, adopting a human-centered approach and emphasizing cooperation with users, as well as evolutionary development strategies and OO software construction.

Evolutionary and cyclical development methods [10] have proved to be best suited for developing software for socio-technical systems, i.e. interactive software. Such methods foster communication between all relevant participants and promote a mutual learning process. Communication and active learning play an important part in training software engineers.

Our view of learning follows Piaget's idea [12,13] of acting towards a goal, collecting experience and critically reflecting on this experience in order to accumulate knowledge that can be used as a basis for new learning. This is a cyclical approach, too. The learning environment should support these activities not only by offering a realistic setting but also by providing the space (scope??), freedom and skilled support needed.

To constantly improve our software development instruction, we are continually assessing our teaching experience. The method adopted here is based on Action Research [2,11]. It involves a cycle of planning, taking and evaluating action, specifying the learning, and diagnosing (*Satz ist schwammig: besser umformulieren!!*). We support our observations by conducting regular surveys and offer help in designing the teaching setting.

3. Experience with Software Engineering Training in Industry

Before describing our experience with software engineering training in industry, we should provide a little background information. The company responsible for the industrial training, it-wps GmbH, is a spin-off of Hamburg University's Faculty of Informatics. Some of the Software Engineering Group's staff members work part-time at it-wps as well, and one of the group's professors is its managing director. The company's work focuses on OO software construction, consulting and training. The following sections describe the didactic aims pursued in our industrial training programs, our experience with time constraints, the organizational structure that has evolved over the years, and our core principles and best practices.

3.1. Didactic Aims

Professional software development involves teamwork. The technical skills of the project team members are important, and substantial effort is normally required to improve the technical knowledge of software development teams. However, a number of soft skills are just as crucial to a project's success: software engineers must be flexible and communicative; they must be used to giving presentations; they must keep their sights on the whole picture; and they must assume responsibility for the project they are part of. These and other soft skills are often neglected when training software engineers. We address this problem by giving equal emphasis to the following three objectives in our training measures:

- imparting sound technical knowledge (e.g. programming languages, design patterns, algorithms, databases, web technologies)
- imparting sound methodological knowledge (e.g. development of quality architectures and designs, software development process models)
- improving soft skills (e.g. presentation techniques, offering and accepting criticism, encouraging a culture of continuous feedback as part of a development process, handling individual and group conflicts, assuming responsibility, heading and coordinating teams, learning to learn)

3.2. Constraints

When training professionals, we had various arrangements with different companies to achieve these didactic aims. The two main variables in all training measures were:

- the time available for the training program
- the skills already possessed by the trainees

If the trainees are novices in OO programming, a training program lasting less than nine weeks will definitely not help them to attain the above aims. In our experience, novices need at least seven months of training to reach the required level. Trainees already familiar with OO techniques can be trained in about five weeks. Here, we attempt to transfer the experience gained in industrial training programs to undergraduate student instruction. In the following sections, then, we refer to our experience with the training of novices.

3.3. Organizational Structure

Over the years, our industrial training of novices has gradually evolved into a three-part program. This is structured as follows, provided sufficient time is available:

- a) in-lab teaching (three and a half months), interleaved with a large number of exercises
- b) miniproject (two weeks), to gain experience of a complete software development process
- c) a real in-house project (three months), conducted by the trainees as a project team

The in-lab teaching phase is conducted in an “intense” classroom setting, as described in [4]. In particular the 3/2-day pattern described in [4] (each week comprising 3 days of information input with exercises and short feedback cycles, followed by 2 “consolidating” days with larger assignments) has proved successful in several completed programs.

The miniproject introduces the concept of teamwork and gives a first impression of what a real full-fledged project is like. Here, the development methodology used is XP [3], with its clear focus on programming, but offering additional communication and feedback mechanisms. The trainer plays both the part of the customer and acts as coach, as addressed in [6].

In preparation for the real in-house project part of the program, we examine all ongoing projects within the company and choose one that is relevant but not critical. By working on an in-house project with real customers, the trainees are able to assume the role of project team members rather than that of mere programmers. They experience typical project problems at different levels: social, technical, organizational, etc. This is where the soft skills come in. The project allows us to demonstrate our core principles and best practices.

3.4. Core Principles and Best Practices

To achieve our didactic aims, we apply several principles and practices. Some are more important during in-lab teaching, others are more relevant during the in-house project, but the trainers must bear them in mind throughout the program. These principles and practices have already been described in [4]: (1) Concepts over API details, (2) Best tools for the tasks, (3) Intense attendance, (4) Learning by doing, (5) Iterative and incremental learning, (6) Permanent reflection, (7) Intense and personal feedback, (8) Soft skills as an explicit topic, (9) Learning environment close to the future job setting, (10) Pair programming.

4. Transferring the Concept

Given the good results obtained when applying our industrial training concept and the growing number of drawbacks we observed in traditional university instruction, we began transferring the concept to a university setting.

Obviously, a university is a totally different environment from industry. However, since university teaching varies from one country to another, we begin by outlining the general situation at German universities. We then go on to describe how we adapted each element of our in-house concept to the academic environment.

4.1. The Teaching Structure at German Universities

At German universities, students have to spend a number of weekly “university hours” (45 minutes) on each course. A one-semester course typically spans 14 weeks. From a student’s perspective, a two-hour lecture involves 90 minutes of class attendance and about the same amount of time for preparation. By contrast, a two-hour tutorial involves 90 minutes of class attendance and some 180 minutes of preparation.

A typical course set-up is a two-hour lecture combined with a two-hour tutorial. Students hear about a topic in the lecture, then go to the tutorial where they are given an assignment on that topic and talk about questions relating to it. During the week, students spend time doing the assignment, either at home or in unsupervised labs. At the next session, they present their work to the tutorial and the tutors collect the students’ completed assignments. Then a new assignment is handed out. After another week, the students get their assignments back with comments from the tutor. Commonly encountered problems are discussed. Students are typically encouraged to work in groups of two to four.

4.2. Studying Informatics in Hamburg

The University of Hamburg is a public university, open to anyone with the German university-entrance qualification (Abitur). The undergraduate program in Informatics begins each year in the fall. There are from 220 to 400 beginners each year, depending on several external factors. Instruction in programming concepts is given in three courses – P1, P2 and P3 – currently covering logical and functional programming (P1), imperative and OO programming (P2), and advanced programming concepts such as concurrent programming, database programming and transactions (P3). Each of these courses takes the form of a general lecture combined with tutorials.

From the teaching perspective, each combined course of lecture and tutorials for approx. 300 students requires a teaching staff of about 15. In addition to a professor, who is responsible for overall course content and lecturing, there is another person in charge of organizing all related formal and conceptual activities (e.g. managing enrollment, writing credit certificates), and some thirteen staff members/graduate students working as tutors in the tutorials. From the students’ perspective, the combination of lecture and tutorial follows the general pattern outlined above.

4.3. Drawbacks of the Traditional Structure

The traditional teaching structure does not work well for programming courses. This is evidenced by the P2 course, which is taught by the Software Engineering Group. When conducting this course, we faced the following problems:

- The old concept required repeated efforts on the part of teaching staff to design assignments so as to counteract the effects of students’ providing answer sheets to students of

the following year. This increased the workload of the tutors, who have traditionally been involved in this design process.

- Correcting students' work entails a substantial workload of ~6 hours/week for each tutor.
- The assignments bear very little relation to real-life projects. The interrelationship between analysis, design and construction in software development can hardly be taught in the form of assignments designed for weekly sessions. Learning targets that go beyond technical questions are hard to teach in this kind of setting.
- Assignments take little or no account of prior knowledge.
- The period of three weeks for presenting the solutions, collecting students' completed assignments and handing back the corrected work is far too long to support a learning process. There is too little individual feedback, which is usually in written form on paper or via e-mail. Tutorial time is normally too limited for detailed feedback.
- Student work groups tend to favor the division of labor rather than teamwork for completing assignments. Overall, there is too much scope for cheating.
- OO programming requires conceptual understanding *and* practical experience.

To summarize: the results of our university course by no means justified the effort involved in preparing and conducting it. Also, in oral examinations many students failed to show sufficient understanding of the core concepts.

4.4. Mapping from Industrial Training to University Education

In the setting described above, it seems reasonable to make use of our experience in industrial training. But the question is: How well can the principles and practices be mapped from one environment to the other, working with the limited resources available (personnel, room space, etc.)?

Bearing in mind the time required to train novices and the special needs of a university environment, we applied our industrial training concept to a number of courses, ranging from undergraduate to graduate classes: an undergraduate lecture with supervised lab work (P2), an undergraduate miniproject, a graduate project and an optional industrial internship. The focus of these courses shifts depending on the maturity of the students and the different settings.

Our many years of experience in the IT field have taught us that programming languages and APIs (Application Programming Interfaces) come and go. It is therefore essential to concentrate on the concepts of programming languages and basic APIs and apply them using an exemplary language. We favor Java as our teaching language. Java is used consistently in all courses conducted by the Software Engineering Group.

The aim of the P2 course (lecture with supervised lab work) is to provide a sound grounding in OO programming. It is therefore crucial that students taking this course acquire a consistent and consolidated picture of this topic. In traditional tutorials, students usually pay little attention and lack the necessary sense of involvement. To address this problem, we conducted P2 with "intense attendance" (3), providing weekly tutorials of 180 minutes in which students actively programmed in supervised labs. This approach encourages "learning by doing" (4) because students have to complete their assignments within the given time frame. In this setting, we are able to give "intense and personal feedback" (7), which allows us to monitor students' progress as well as the problems they face.

By asking students to work in pairs, we encourage them to develop their interpersonal skills ("pair programming" (10)). They are urged to ask questions and answer them; they have to justify their decisions; and they learn to respect each other's actions. This encourages "permanent reflection" (6) on the topic.

We follow the rule “concepts over API details” (1) by setting assignments that do not deal with specifics of an API and by encouraging students to acquire skills in searching for the relevant information using, for example, the API documentation.

The first eight weeks (phase one of the course) are devoted to basic programming concepts. Objects First [1] is used as the teaching method. The last six weeks (phase two) cover a single topic involving more complex development activity. Here, students gain experience with programming problems relating to complexity.

Following the principle “Best tools for the tasks” (2), the BlueJ [8] environment is our preferred choice for teaching the basics of an OO programming language. In the second phase, we introduce Eclipse [9] as an IDE in order to familiarize students with a professional tool. Being able to choose the IDE freely is an advantage compared to industrial training programs, where we normally have to use the company’s in-house tool without the opportunity to assess its suitability.

In the undergraduate miniproject, conducted during the three-month break between students’ second and third semesters, we intensify the programming work by getting the students to tackle a single project for five whole days within one week. Teamwork is required because the task is too large for individuals or pairs to perform alone. Three or four project teams, each comprising 10 to 14 members, work independently on the same task. 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 provides experience with group dynamics. In addition, we consider “soft skills as an explicit topic” (8) by getting the teams to give presentations of their final results in front of the whole course group.

The overall concept of undergraduate teaching stresses “iterative and incremental learning” (5). Each assignment builds on the previous one; describing one concept requires an understanding of a lower-level concept (e.g. reference and polymorphism). Moreover, students are urged to obtain the necessary information as required.

Some didactic aims obviously cannot be met in the undergraduate courses, e.g. the “learning environment close to the future job setting” (9). To achieve these, we offer projects and internships at the graduate level with industrial partners. The focus of the graduate software engineering project is to provide a complex software development activity spanning two semesters. We have discussed this concept and some aspects of our experience with it in [7]. The industrial internship is a voluntary activity that offers committed students the opportunity to experience real-life software development situations and cannot be dealt with in detail here.

5. Evaluation of the First Two Years

Graduate projects based on OO have formed an integral part of our graduate teaching programs since 1996. We have been conducting the undergraduate miniproject 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. Our experience with projects is, then, well established and assessed [7, 6], whereas the P2 concept with supervised lab work is a fairly new experiment [4, 5].

During both semester breaks following the new P2 course, we conducted and evaluated three surveys: one for students who had completed the course successfully (grade: pass or better), one for students who had failed the course, and one for the instructors. We conducted anonymous surveys in which the participants were able to tick preformulated statements as well as providing individual feedback.

5.1. Student Feedback

Students who successfully completed the course gave very positive feedback. 92% did NOT tick the statement “I would have preferred a traditional course with tutorials and as-

signments". 82% ticked the statement "The lab classes were great 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 to do their job". In the free-text answers, most students complained that there was not enough time for individual feedback and that there should be a better instructor/student ratio.

The feedback from students who failed the course showed that the new concept had little impact on the failure ratio. Only 16% blamed their failure on the new concept, most attributing it to personal problems or external pressure. One student who had successfully completed the course the previous year took it again because he was curious about the new approach. He stated that he liked it better than the old concept.

5.2. Instructor Feedback

We were particularly interested in changes in the instructors' workload due to the new concept. About 70% of respondents stated that the new concept took less or the same amount of time as the old concept. 30% refrained from answering on the grounds that they had no prior teaching experience which would enable them to draw a comparison. The majority (again about 70%) said that their personal stress levels during contact hours were acceptable or lower than before, while the other 30% found the new concept more strenuous.

In the first year, most instructors criticized what they saw as the too low instructor/student ratio. Towards the end of lab sessions, there was often not enough time to assess students' work, which was explicitly desired by the students.

70% of the instructors considered the new concept a substantial improvement, 30% saw no difference, and none regarded it as inferior to the old approach.

5.3. General Observations

Most of the (few) problems encountered in the first year were able to be avoided in the second year:

- The assignments merely required polishing in the second year, which lessened the instructors' workload substantially.
- The faculty funded more instructors, leading to an improved instructor/student ratio.

55% of students dropped out of the old P2 course in 2002. After the new concept had been introduced, the figure fell to 50% in 2003, and further to 22% in 2004. Another positive result of the new concept's introduction was that the students who dropped out did so early on in the semester. We thus achieved our objective of enabling students to determine early on in the course whether they considered themselves capable of attaining the learning goals or not.

One of our observations concerning the old concept, namely that the interrelationship between analysis, design and construction in software development can hardly be taught in assignments designed for weekly sessions, still applies to the new concept. Students are confronted with this issue for the first time in the miniproject, but already in a team context. What is missing is a setting somewhere in between, where individuals experience how a concrete, real-life problem can be solved by software that is designed and implemented following analysis of the problem.

We observed that instructors from outside the Software Engineering Group with no experience of industrial training programs had difficulties coping with the intensity of the lab work. Most of them found supervising two three-hour lab sessions quite stressful; four three-hour lab sessions stretched them to their limits. By contrast, teaching staff that were used to working full-time in industrial training programs were less affected by this.

Instructors supervising lab work obviously need extensive programming experience, in our setting with imperative and OO programming in particular. Since experience with these paradigms still cannot be taken for granted, instructors have to be carefully selected. Some graduate student instructors are better suited to the job than some of the university teaching staff that grew up programming in a different paradigm.

6. Discussion

On the whole, it can be said that our experience with industrial training programs has so far proved easily transferable to a university setting. There are, however, some problems that still need to be addressed:

- the development of soft skills for presenting small designs
- occasional large discrepancies between the respective skill levels of pair members
- the fact that instructors' skills are not necessarily tailored to the topic being taught
- organizational problems (tutors and students have difficulty attending courses with atypical time frames, lack of time to review assignments at the end of a session)

The focus of the P2 course is on core programming practices, little attention being given to soft skills (except for feedback within programming pairs). The old concept had students presenting their solutions in tutorials of around 20 people. This was abandoned in favor of improving programming skills, but in future settings oral presentations should be revived.

In the near future, the University of Hamburg's Faculty of Informatics will be changing its curriculum to bring it more in line with the Anglo-Saxon model (Bachelor's programs). The current plan involves doing away with functional and logic programming in the first year and devoting more time to imperative and OO programming. We take a positive view of this shift in focus because it allows us to give more detailed treatment to a number of important points that time constraints have so far prevented us from covering.

7. Conclusion

In this paper, we have reported on our experience with transferring successful industrial training concepts to university education. This time, instead of focusing on one specific course, we have tried to present the big picture: from first-year programming courses to graduate projects to industrial internships. We feel that all computer-related education – whether it be in computer science, software engineering or information systems – should give proper attention to OO analysis, design and programming, along with the soft skills needed for presenting and explaining the resulting software artifacts. We believe that industrial training can be an important force for innovation in university education if teaching staff are able to acquire experience in both worlds.

8. 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 (CSEET 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: M. Marchesi, G. Succi (eds.), *Extreme Programming and Agile Processes in Software Engineering*, Proceedings XP 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 (eds.), Workshop of the German Chapter of the ACM and the Gesellschaft für Informatik (GI), pp. 9-20, 1999.
- [8] BlueJ. www.bluej.org (last visited January 4, 2005).
- [9] Eclipse. www.eclipse.org (last visited January 4, 2005).
- [10] Floyd, C., F.-M. Reisin, and G. Schmidt. Steps to software development with users. In C. Ghezzi and J.A. McDermid (eds.), *ESEC89, Number 387 in Lecture Notes in Computer Science*, pp. 48-64, Springer-Verlag, Berlin, 1989.
- [11] Johnson, B. Teacher-As-Researcher. ERIC Digest. ERIC Clearinghouse on Teacher Education Washington DC. ED355205. 1993
- [12] Piaget's theory. In P. Mussen (ed.) *Handbook of child psychology*, Vol.1. New York: Wiley, 1983.
- [13] *Studies in reflecting abstraction*. Hove: Psychology Press, 2000.