Software Engineering Education: The Synergy of Combined Research and Teaching

Matthias Gehrke, Holger Giese, Ekkart Kindler, Jörg Niere,
Wilhelm Schäfer, Jörg P. Wadsack, Robert Wagner and Lothar Wendehals
Software Engineering Group, Department of Computer Science,
University of Paderborn, Warburger Straße 100, D-33098 Paderborn, Germany
[mgehrke, hg, kindler, nierej, wilhelm, maroc, wag25, lowende]@upb.de

Abstract

Teaching software engineering is a sumptuous task. In particular, practical software engineering courses require a lot of experienced teaching staff, who are, as a result, kept away from research projects.

In this paper, we report on a software engineering course using a large interdisciplinary project from current research. This way, the students were challenged by an interesting industrial strength task. In turn, we obtained students who were well-skilled for joining the research projects. What is more, the software developed during the software engineering course immediately contributed to the research project. Altogether, the additional expenses spent on teaching did not only improve the quality of teaching, but also paid back to the research project.

1 Introduction

Teaching practical software engineering is one of the most challenging tasks of computer science education. In particular, it requires a software project to be defined that is of reasonable size, but at the same time solvable within a university term by a team of about ten students. Often, the software projects used are too small, for the extent that students are tempted to immediately begin working on the implementation. They do not see the necessity for an analysis and a design phase, for project management, and for version management. In addition, the projects are too small for training the social skills of the students. Moreover, practical software engineering courses require a lot of staff for teaching and coaching, for technical support, and for organization.

In this paper, we report on the experience of using a software project from current research in software engineering education. This is very much in the spirit of Wilhelm von Humboldt's Ideal of Education [UN93]. Through the tight coupling of teaching and research, the investment in practical software engineering courses pays back to software engineering research.

1.1 Two problems

By combining research and teaching, we can solve two problems with a single solution. One problem comes from software engineering education, the other from software engineering research.

In software engineering education, one of the main problems is the definition of an industrial strength software project, i.e., a project that cannot be solved by one or two students on their own, a project that incorporates existing and evolving third party software without any documentation, a project with changing requirements, and a project that has not got a clear outcome right from the beginning. Moreover, the project should be interesting for the students and challenging by allowing the use of state-of-the-art technology. On the other hand, the project should not be too large, such that a team of students will be able to finish it within a university term.

In software engineering research, one of the main problems is recruiting new students for research projects. These students should be sufficiently skilled in software engineering, should have some experience in the respective research area and application domain, and should be highly motivated.

1.2 One solution

Both problems have a common solution: using software projects from research projects in courses in software engineering. Moreover, the teaching staff can then be supplemented with the experienced PhD students of the project which benefit from the course outcome. Of

course, this is only the basic idea. For example, not all research projects are good candidates for software engineering courses. Moreover, many conceptual, technical and organizational problems remain to be solved. Some of them do not only concern the software engineering course itself, but do concern the complete computer science program. These issues will be discussed in Sections 2, 4 and 5.

1.3 The bonus

An immediate benefit of using a research project in a software engineering course is the number of independent teams working on the same project. If there are enough well-motivated teams, we can be sure that at least some teams will come up with really good results. In the end, we only have to choose the best one, which can then be used in the research project and can be further developed in next year's course. We call this concept *competitive software engineering*, which seems to be a good concept if human resources are for free. Clearly, in industry, human resources are not for free. But at universities, the situation is different. Why shouldn't university research benefit from this? At the University of Paderborn, we had about 200 students working in 18 teams, some of which came up with excellent results.

Usually, the results of a software engineering course are simply thrown into the bin. If we are interested in using the results in a research project, we need to spend some extra effort in order to guarantee quality, compatibility, and maintainability of the results (at least of the winner of the competition). We will report on the extra measures taken to achieve this goal in Sect. 5. We must admit that we underestimated this extra effort in our first course. Nevertheless, the extra effort pays back when considering the overall advantage.

In the following, we will report on the details of our software engineering course. In Sect. 2, we will discuss the computer science program at the University of Paderborn, and we will briefly summarize several years of experience with software engineering courses that did not use a project from research. In Sect. 3, we will discuss the associated research project. In Sect. 4 and 5, we will discuss the concrete software project and the organization of the course. A summary of our experiences and some conclusions follow in Sect. 6 and 7.

2 Context and content

Clearly, software engineering cannot be taught in a single software engineering course. For proper software engineering education, the complete computer science program must be adjusted to this goal. In this section, we briefly discuss the computer science program at the University of Paderborn, our teaching goals, and the key issues of earlier software engineering courses.

2.1 Computer science program

At the University of Paderborn, software engineering is a basic thread of the computer science program. Before attending the practical software engineering course, students must attend a programming course and a software technology course. In these courses, they are provided with basic programming skills in Java and with basic knowledge in software engineering technology such as life-cycle models and UML. Up to this point, however, the students' practical experience in programming as well as in using UML is restricted to small examples. During the practical software engineering course in the end of their second year, students are faced with a larger software project for the first time.

In order continue with software engineering after the software engineering course, the University of Paderborn additionally requires an internship in industry and participation in a one year project, which we call a 'project group'. The internship in industry provides more insight into industrial software development. Since the industrial partners are also partners in our research projects, students stay in contact with research during their internship. In a project group, once more, a group of about ten students works on a software project. Typically, the software project comes from the research of one or two research assistants or PhD students, who supervise this group. In the project group, the students do research and can continue to train their software engineering skills. This time, however, they are faced with much tighter conditions, because each individual group must provide a useful result. Therefore, the prior practical software engineering course is essential to the success of the project groups. The practical software engineering course has another useful effect on the project groups: during the undergraduate course in software engineering, the students get to know their supervisors, who, in most cases, are PhD students or research assistants doing their PhD. On the other hand, the supervisors get to know their students. Therefore, the supervisors can choose the best skilled students for their research projects and encourage them to join their project groups, or to start a bachelor or masters thesis in their project.

2.2 Teaching goals

The main purpose of the practical software engineering course is to demonstrate the necessity of using software engineering techniques for producing software. Students should experience that jumping straight into programming (hacking) won't work anymore when the projects become larger. The students should see

the necessity of all phases of software engineering [Cus00]. In particular, this applies to requirements engineering, design, test and installation (system integration) phases. Moreover, students should get a feeling for the documents that must be produced in the different phases. They should know which information must be contained in a requirements document and a design document.

Students should also experience that developing software in a team is completely different from programming on their own. They should see the necessity for project management, and they should train their collaboration and communication skills. They should experience the time pressure imposed by a customer in combination with real-life adversities such as changing requirements or changing deadlines. Finally, they should learn how to present their work to a customer and to communicate with him.

Another important teaching goal is reverse engineering. Since most real-life software engineering projects use some existing software without much documentation, students must know how to tackle this problem [Bot01, SSvdW99].

2.3 Earlier courses: key issues

In [GGN+02], we reported on our earlier software engineering courses, which aim at the above teaching goals. Here, we summarize the key issues of these courses. As proposed by a number of software engineering education papers [MR99, AL00, Bot01, Vau00, SSvdW99, BC02], we used industrial-strength projects in our courses. To motivate and challenge the students, the projects were structured in such a way that it was possible to have a contest at the end.

Concerning project management, the teams had to meet several deadlines at which certain documents had to be delivered. Each student was assigned the responsibility for a specific task, which he had to coordinate in his group. In order to coordinate work on the same documents, students had to use the Concurrent Versions System [CVS]. To simulate real-world adversities, we applied some of Dawson's dirty tricks [Daw00].

Moreover, the teams had to prepare a talk on their requirements document. In the end, they had to prepare a Web site containing a complete documentation of their project and from where the software could be downloaded.

3 Research project

Instead of letting the students program just another standard electronic timetable or database application, we looked for a more motivating application example. Our current research in the field of embedded real-time systems provided a suitable candidate. When complex concurrent software is developed in this application domain, fundamental software engineering techniques such as UML class diagrams, sequence diagrams and statecharts are required to analyze and design the system. In contrast, for the above mentioned standard application examples, the student projects would often become a pure academic exercise, as appropriate commercial database application design tools exist today, which simplify the task at hand greatly. The mechanical and physical aspects of an embedded system also result in a clear and self-contained problem where the need for appropriate software quality and safety is more obvious than for standard example applications.

As the underlying research project for the software engineering course, we chose the new collaborative research center 614 of the German Research Council (DFG), titled "Self-optimizing Concepts and Structures in mechanical Engineering"[1]. The general vision of this collaborative research center is mechanical products with inherent intelligence, which can react autonomously and flexibly to changing environment and operation conditions. As a concrete example, an already existing railcab research project[2] has been chosen, which aims at using a passive track system with intelligent shuttles that operate individually and make independent and decentralized operational decisions. This autonomous shuttle concept allows the gap between short- and long-distance traffic to be bridged, and will bring considerable increase in traveling comfort (e.g., no changing of trains) and higher passenger flexibility. The infrastructure is built by satellite-supported positioning and a cellular-phone network for shuttles, to enable communication between shuttles and stationary installations. The modular railway system further combines sophisticated undercarriages, with the advantages of new actuation techniques, as employed in the Transrapid[3], while using the existing railway tracks.

The main elements of the proposed intelligent shuttle system are depicted in Fig. 1. The proposed scenario results in a number of complex logistics optimization problems. To reduce the energy consumption due to air resistance, appropriate coordination between the shuttles is required so that convoys can be built. To reduce costs, each shuttle is built in a modular fashion. The most ambitious module is the complex undercarriage, combining a linear motor with a wear-free guiding. The collaborative research center will address self-optimization for each of these different elements.

The logistics of the system include the early detection and bypassing of congestions. The booking system will

[1] http://www.sfb614.de
[2] http://www-nbp.upb.de/en/index.html
[3] http://www.transrapid.de/en/index.html

manage demands for transport of passengers and cargo. It will support client-specific rates to optimize shuttle operation. When self-optimization techniques are additionally employed, significant improvement for the utilization of the shuttle system can be expected.

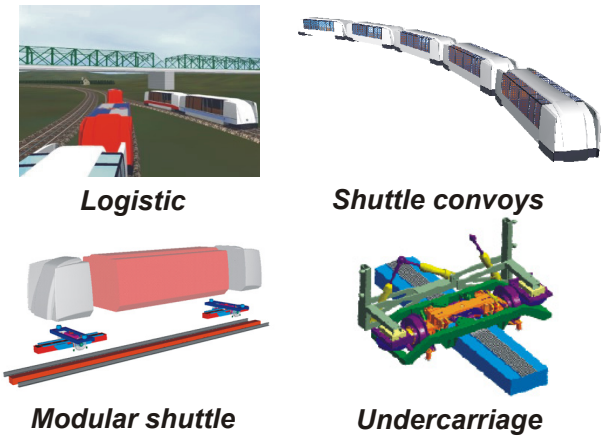

Fig. 1: Elements of the Intelligent Shuttle System

While each shuttle operates autonomously, the software-based coordination between them will ensure system safety as well as optimize their energy consumption. To reduce air resistance, the shuttles have to drive as convoys with minimal distance between them. The coordination between the speed control units of the shuttles then becomes a safety-critical aspect, and results in a number of hard real-time constraints, which have to be addressed when building the distributed control software. Self-optimization can be used to optimize the average-case performance of the system, while its worst-case behavior must be considered to ensure the required real-time behavior.

The modular shuttle construction planned, has further considerable potential for cost reduction through the combination of standardized system modules with the predicted large number of required units. Therefore, each shuttle is built from modules. When composing the different modules, a hierarchical structure of self-optimizing control software is employed to ensure that sub-modules optimize their behavior with respect to the goals of their overall module.

Each shuttle is equipped with a linear motor which runs on existing tracks. An intelligent undercarriage provides considerably higher ride comfort. The wheels of the shuttle are only used for guiding and braking and therefore the wear of wheels is considerably reduced. The linear motor principle allows contact-free power transmission into the shuttle and thus power transmission by power lines or rails is no longer necessary. These innovative technical solutions further require sophisticated shuttle software that optimizes the shuttle behavior, also taking energy constraints into account.

The quality of complex mechatronic products, today and in the future, will depend crucially on the high quality of their software components. When self-optimization comes into play, the role of software and its flexibility becomes even more prominent. Within this collaborative research center, the software responsible for the self-optimization is considered to be a multi-agent system (MAS). The contribution of our software engineering team to this research project will be the development and adjustment of fundamental software engineering techniques for the collaborative research center. The resulting design technique will support the following features:

- support for the design of self-optimizing MAS
- support for the design for hard real-time systems
- integration of control into complex distributed and knowledge-based software
- pattern-based design with partial synthesis of the component behavior
- compositional verification of safety-critical system properties for the resulting real-time software
- an *environment simulation* [Gar99] for embedding and testing the real-time shuttle software that can be later combined with more detailed simulations of the physical shuttle behavior

To achieve all these goals, a combination of object-oriented and component-based techniques with the multi-agent paradigm will be developed, where each agent is operating autonomously and trying to fulfill its own goals pro-actively using the knowledge base following the principle of self-optimization.

4 Transformation to a student project

Of course, such a long-term project as described above is by far too ambitious and too complex for a software engineering course. However, presenting the context of such a project to students makes them aware of interesting and challenging research problems. The possibility of developing software which could be partly re-used in the context of a large research project increases their interests and active participation in the project significantly.

The general idea for stripping down the project to a manageable and adequate size for a one-term student project is to drop any (hard) real-time requirements and any sophisticated optimization algorithms, and to keep the model of the shuttle very simple.

In more detail, the task of each team is to (1) develop a (sophisticated) 2D graphical user interface, (2) a smart algorithm which optimizes the profit of a single shuttle, and (3) an editor which supports the construction of maps consisting mainly of tracks and stations.

As a basis for an environment simulation, a simulation kernel was developed before the project started and given to the students (see next section for more details). This kernel supports the concurrent execution of an arbitrary number of shuttles and randomly generates offers to be taken by an individual shuttle. Each offer consists of a start and final destination together with a deadline, a virtual load, and a value which the shuttle earns if it arrives at the final destination before the deadline expires. If the shuttle arrives after the deadline, it pays a penalty. Using tracks and stations when delivering the virtual load of the shuttle as described in the offer, requires a shuttle to pay fees.

Each shuttle can now individually optimize its strategy based on its current position at a particular station, its current profit and the known costs for each track and station to be passed when delivering the load. Some more details of the rules of the game are omitted here. Taking them into account gives students the opportunity to explore very different and more or less sophisticated optimization strategies to maximize the profit of a single shuttle.

The graphical user interface displays a map of tracks and stations and illustrates the moves and positions of all shuttles as they are delivering their loads and moving to their final destinations. Of course, creativity has almost no limits when designing the layout of such a map, as, for example, the layout of stations, of shuttles and of the track system can became very sophisticated.

Finally, the editor supports the construction of a single map consisting of stations and tracks and their respective costs. It controls the correct construction of a map as, for example, there is no station which is not either the start or end of a track.

The competitive nature of the course is underlined by a tournament at the end of the term. All shuttles from all teams with their independently produced algorithms run concurrently on a map for a certain period of time. When time expires, the individual profits identify the winner, runner-up etc. of that round. A number of rounds are played to determine an overall winner. Different maps are constructed using the above mentioned editors.

In fact, the individual algorithm of the team's shuttle becomes each team's protected knowledge which could bring them the competitive advantage during the tournament. In addition, the sophistication of the map display and editor functionality is further input to the final grading of the project.

In general, the student project requires substantial knowledge in writing Java-programs for soft real-time systems and using libraries for building sophisticated (2D) user interfaces. It does not include using any hardware-oriented programming interfaces or obeying constraints which originate in the continuous nature of the research project. For example, the complex control theory which is needed to control the building of shuttle convoys is not addressed in the student project.

5 Key issues of the development process

As explained above, we have substantial interest in reusing results and especially software which is produced by the course participants. This requires a very well-defined and strictly monitored process, because this course is the first exposure to a reasonably-sized software project for most students. We focus here on the key issues in addition to the process definition as described in section 2.

5.1 Strict project management

Firstly we distinguished clearly between the team supervisors who was either an experienced PhD student or in some cases a teaching assistant, and the group leader who was a member of the student team. The supervisor had to interfere if the work went into the wrong direction and give general advice. The group leader or project manager was really in charge of coordinating and controlling the work of the whole group. This sharp distinction was necessary to ensure that supervisors, who are involved in the large research project in most cases, mainly play the role of customers and do not become team members at least in an unofficial way.

Secondly, the project plan consisted of hard deadlines. At every deadline, the teams had to deliver their results of the different project phases to the project manager who forwarded them to a central course-wide management authority (which later on turned to be the same person as the one who managed the hotline, see below). Missed deadlines implied a decrease of the team's final grade. The state of the deliveries of all teams was presented on a website accessible to all course members. This approach raised the peer-group pressure and made students sensitive to the group dynamics and responsibilities.

Thirdly, to ensure a certain quality of the deliveries, we provided content lists as well as a defined structure of each document. In addition, we made excellently graded document examples from former years available for the students.

Fourth, in order to raise the quality of the individual team product, two customer presentations took place in the middle and near to the end of the project. Both presentations were presented to independent persons, i.e. supervisors who were not the supervisor of the group who gave the presentation. This ensured real independence and tough questions and discussions as is the case when a industrial presentation is made to a customer. The presentations were graded as part of the final grade. The first presentation included mainly the requirements document. This presentation was crucial to the teams,

because they had to reflect about their ideas and approaches for performing a clear and comprehensible presentation. The second presentation was used to show the developed application which then became the software for the final tournament as mentioned above.

Finally, we introduced time estimation and time control throughout the project to make sure that (1) students understand software also as an industrial product which requires a cost/benefit analysis and (2) to control that our requirements do not overload students. The estimations had to be delivered after finishing the reengineering phase, i.e. after the fourth week. The estimation included the time for design, presentation, documentation, implementation and tests. During the whole course we asked for weekly time sheets, where each team member's time effort was reported. The time estimations and reports were not graded.

5.2 Using existing software

The soft real-time simulation kernel was developed upfront by senior students under the supervision of an assistant professor. This software also included a visualisation component and a shuttle, which was able to move on a sample track system, to bid for orders and to perform an order after its assignment. This software was used by the students to understand the basic logic of the whole program and it served as the basis for the reverse engineering exercise at the beginning of the project.

In addition to providing a simulation kernel for the project and relieving students from programming a complex complete concurrent system, we had some other goals. One goal was testing the stability and performance of such a kernel, which was achieved by using it in the tournament where 18 shuttles ran for several hours. A second goal was to evaluate different techniques for building the kernel and to reveal problems in the system architecture which was achieved by getting a lot of feedback from the students using the kernel.

The kernel development was only completed during the course which simulated "real-life software development" where interfaces and sometimes even functionality of a piece of reused software may change quite surprisingly and suddenly.

The independent development of the kernel enforced a way the teams can contact the kernel developers. Therefore we installed a central hotline to coordinate bug reports and fixes as well as questions and improvements. The hotline was available via email and phone. During the course, managing the hotline was done by one person only, who was also highly involved in the development of the kernel. This reduced the answer time of questions, because the hotline employee had not to contact the developers so often.

6 Lessons learned

As we did this experiment the first time, namely to introduce research into an undergraduate software engineering course, we learned a lot, especially concerning student motivation, the results of time estimation and control, and the exploitation of our hotline.
We also say a few words about the groupware aspects in each group.

6.1 Motivation

We observed that the context of a large research project had motivated the students to perform such a large, difficult and time expensive course. The students realized that their outcome will be part of the research activities of the university and will not be thrown away afterwards. The high motivation of the students caused them to incorporate more features in their product than they had to. Examples are the use of 3D instead of 2D visualization techniques, which were postulated, cf. Fig. 2. Some teams included minimized maps to present an overview of the whole map and the editing functionalities were very impressive and easy to handle. In addition some teams developed also some marketing strategies such as a product flyer for their presentations or merchandising. During the tournament, some teams showed also high team spirit by wearing shirts with their team logo. Teams from previous years and even fellow students came to the tournament. These and several more examples show that the right task has a very strong impact on the motivation of the students.

6.2 Time estimation

Due to the lack of experience of the students the time estimations were mostly imprecise, but they were imprecise in a similar way. On the one hand most of them estimated the effort for design, presentations, documentation and tests too high. On the other hand they underestimated the effort for implementation. The time for design was particularly overestimated and the design phase was even disregarded by most teams. The need for a good design is hard to teach in a project like this. In large industrial projects, the design should be seen as a contract between two parties: one which produces the design and another that uses the design for the implementation. These two parties depend on each other. This concept is not applicable to our course, as this course should introduce the students into the whole software development process, so the teams can not be split into designers and programmers.

The team's estimations of the overall projects ranged between 480 and 2,200 hours spanning over 15 weeks with a team size of 11 students. The actual used effort

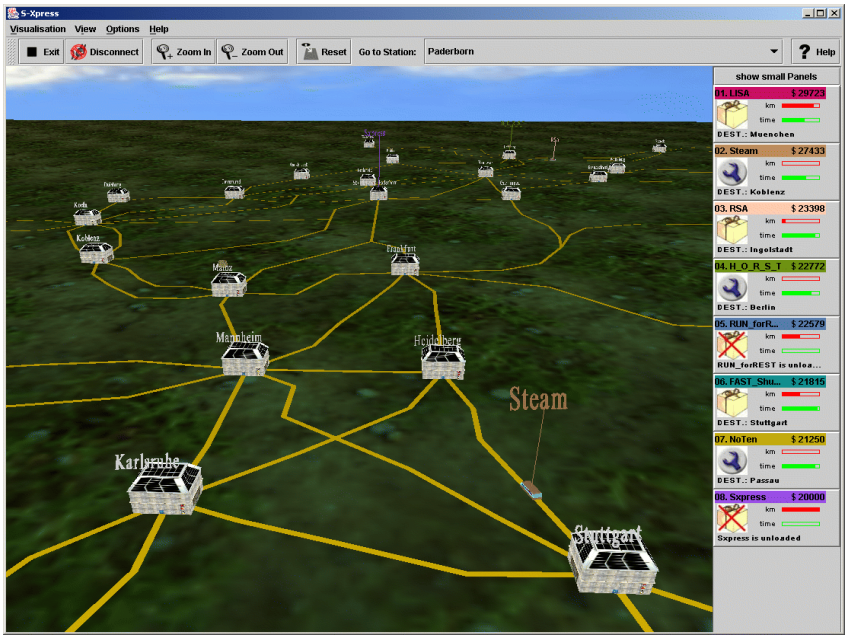

Fig. 2: 3D visualization component

ranged between 640 and 1,930. This shows the right dimension of the project, as we expected not more than 12 hours per student and week, which is 1,980 hours in 15 weeks.

6.3 Groupware

Working with configuration and versioning tools is a major learning step for our students. In our experience, it is hard to teach the benefits of a versioning system in a lecture. Students must use a versioning tool in a real team project in order to understand its value. When introducing the use of a versioning tool, students have many objections and doubts that versioning works properly and supports valuable team support. The first steps with the versioning tool are often confusing and considered as overhead. Although there are few students who didn't use the versioning system after a few weeks most of the students couldn't imagine how to do teamwork without such a system. Some students with experience in teamwork not employing a versioning system are very content with the great relief from coordination efforts. They usually start to promote versioning systems to other project and even to companies they are working for.

6.4 Hotline

Industrial practice includes changing requirements over a project's duration. As mentioned in the last section, the kernel evolved during the project, e.g. adding events, changing protocols and also offering new abilities to get information from the kernel. Consequently the teams had to adapt their software over the whole project. To coordinate and manage the changes and resulting questions, a hotline was installed as mentioned above.

The hotline person reported at the beginning of the project that the job sounded easy, e.g. collecting deliveries from all teams and archiving them for the other supervisors, or selecting and sorting the inquiries and bug reports of the kernel. During the project, the time spent on the hotline activities raised. For example, the hotline was responsible for the installation of each application shown at the final product presentation. The coordination, installation and fine tuning of the final tournament was also hard work, because some teams had not provided their shuttles to the other teams and thus the integration tests were not complete. The hardware environment was different to the test environments, which was caused by some problems with 3D-graphic-accelerator libraries and drivers. On top of this the hotline was a drop-in centre for all kinds of questions concerning the project, e.g. questions about terms, grading, and sometimes also social problems. In the end, the hotline job was not simply a hotline but entailed coordination of the whole project.

During the project we observed that the hotline job was more work than previously expected, but afterwards we agreed that a different management would have failed. A central management is crucial for the success of integrating research parts in software engineering courses.

Finally, our experience of four years running this course indicates that a group size of 10 - 11 members is pretty optimal. It gives a reasonable size to practice teamwork and, in addition, this size compared with

smaller sizes in the past, results in having more technically experienced and skilled persons in Java-programming in the team, i.e. students who already programmed a lot more than was required by the introductory courses as mentioned in section 2.

6.5 Teaching Assistants

For a course with 18 teams it is very difficult to get sufficient suitable supervisors with technical as well as social skills.

We recruited our supervisors from two groups. One group consisted of PhD students (research assistants) with a lot of experience. The other, larger group of supervisors consisted of teaching assistants (TA). These TA's usually did not have enough knowledge to supervise a team. Therefore, one important consideration in the selection of these TA's is that they have participated in the course (as students) themselves. With such a background they know the main problems which may occur and can handle difficult situations more easily. Particularly the social problems are solved faster and more effectively. Therefore, if there is a good supervisor looking after a team, fewer problems will arise and the students will be much more content. Therefore we suggest paying a lot of attention to the selection of the teaching assistants and their technical and social skills.

6.6 Grading

In a course with 18 student teams and 15 supervisors it is hard to grade the teams in an objective way and it took several hours to consolidate on a marking scheme. This year, the strict deadlines, the provided structures for delivered documents and presentations allowed us to compare the team's results more easily. The consolidation time was shorter and grading was easier, but we observed that the concrete grading was not transparent enough to the students from the start, which is something to be improved on over the next years.

7 Conclusions

In conclusion, we can report that we were successful in motivating and interesting students in research work. We got many positive reactions at the end of the project and some students asked us for a bachelor thesis and even more are now teaching assistants in our research group. We succeeded not only in reaching the teaching goals, but also raised the quality of education because of better support of the students. They learned what problems and challenges arise in big software projects and how they can be handled.

We look forward to getting more feedback from a survey we will perform in a few weeks, and also hope to recruit more students for bachelor, master and PhD theses.

Acknowledgement

We thank all teaching assistants and students which were involved in this year's software engineering project.

References

[AL00] J. H. Andrews and H. L. Lutfiyya. *Experience Report: A Software Maintenance Project Course*. In Proc. of the 13th Conference on Software Engineering Education and Training, Austin, Texas USA. IEEE Computer Society Press, March 2000.

[BC02] M.B. Blake and T. Cornett. *Teaching an Object-Oriented Software Development Lifecycle in Undergraduate Software Engineering Education*. In Proc. of the 15th Conference on Software Engineering Education and Training, Covington, Kentucky, USA. IEEE Computer Society Press, February 2002.

[Bot01] K. Bothe. *Reverse Engineering: the Challenge of Large-Scale Real-World Educational Projects*. In Proc. of the 14th Conference on Software Engineering Education and Training, Charlotte, North Carolina USA. IEEE Computer Society Press, February 2001.

[Cus00] J. Cusick. *Lessons Learned from Teaching Software Engineering to Adult Students*. In Proc. of the 13th Conference on Software Engineering Education and Training, Austin, Texas USA. IEEE Computer Society Press, March 2000.

[CVS] CVS. *Concurrent Versions System - The open standard for version control*. http://www.cvshome.org/.

[Daw00] R. Dawson. *Twenty Dirty Tricks to Train Software Engineers*. In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, pages 209–218. ACM Press, 2000.

[Gar99] Stewart Gardiner (editor). *Testing Safety-Related Software: A Practical Handbook*. Springer-Verlag, 1999.

[GGN+02] M. Gehrke, H. Giese, U.A. Nickel, J. Niere, M. Tichy, J.P. Wadsack, and A. Zündorf, *Reporting about Industrial Strength Software Engineering Courses for Undergraduates*. In Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA, pages. 395–405, May 2002.

[MR99] W.W. McMillan and S. Rajaprabhakaran. *What Leading Practitioners Say Should Be Emphasized in Students' Software Engineering Projects*. In Proc. of the 12th Conference on Software Engineering

Education and Training, New Orleans, Louisiana USA, pages 177–185. IEEE Computer Society Press, March 1999.

[SSvdW99] K. Sikkel, T. A. M. Spil, and R. L. W. van de Weg. *Replacing a Hospital Information System: an Example of a Real- World Case Study*. In Proc. of the 12th Conference on Software Engineering Education and Training, New Orleans, Louisiana USA. IEEE Computer Society Press, March 1999.

[UN93] UNESCO, International Bureau of Education. *Wilhelm von Humboldt*. Prospects: the quarterly review of comparative education, vol. XXIII, no. 3/4, pp. 613–623, 1993.

[Vau00] R. B. Vaughn. *A Report on Transfer of Software Engineering to the Classroom Environment*. In Proc. of the 13th Conference on Software Engineering Education and Training, Austin, Texas USA. IEEE Computer Society Press, March 2000.