

# A Practical Approach of Teaching Software Engineering

Michael Gnatz, Leonid Kof, Franz Prilmeier, Tilman Seifert

Institut für Informatik  
Technische Universität München  
85748 Garching

{gnatzm, kof, prilmeie, seifert}@in.tum.de

## Abstract

*In today's software industry a software engineer is not only expected to successfully cope with technical challenges, but also to deal with non-technical issues arising from difficult project situations. These issues typically include understanding the customer's domain and requirements, working in a team, organizing the division of work, and coping with time pressure and hard deadlines. Thus, in our opinion teaching Software Engineering (SE) not only requires studying theory using text books, but also providing students with the experience of typical non-technical issues in a software project. This article reports experiences with the concept of a course focusing on providing practical know-how.*

## 1: Introduction

This article describes a practical approach of teaching SE know-how at the Technical University of Munich in a course called "Software Engineering Project" (SEP) conducted by the authors during summer term 2002.

### 1.1: Motivation

The major objective of the SEP is to provide students with the experience of an industry-like software development project. A typical software project is challenging in many ways: A new domain must be understood, "maybe" requirements are inconsistent and need to be clarified with the customer; the technology used might be new to team members, so they must learn and gain experience using that particular technology. Deadlines might be set a bit too optimistically, and developers need to solve unexpected technical problems. Besides all this, social aspects are to be considered as well: How do team members communicate with each other, how do they like each other's working style, habits, and so on.

There are many SE text books that describe all these difficulties and challenges; some of them like [3] focus more on design aspects, while others like [8] emphasize project management issues. But the only way that really prepares anyone for doing successful project work is learning by doing. From a university's perspective it is a challenging task to find a teaching concept for providing students with practical know-how regarding project work.

## 1.2: Project Goals

One major goal is to teach students what it means to work in a team and how to go through the whole project life cycle with all phases from requirements engineering to delivery. The second goal is the development of a correctly working and useful piece of software which serves at least as a prototype. This software should be well documented and maintainable. During the course of 2002 the tool prototype APE, which is explained in section 3, was developed.

About half of the students had already heard the lectures on SE and knew about typical problems and methods useful in solving them. They knew in theory what to do. Our project showed once more how important it is not only to theorize about developing software, but to actually do it. Only when time becomes short and deadlines come closer, everyone (including students and supervisors) realize the importance and difficulties of social communication and stable interface definitions on the technical side.

## 1.3: Overview

In section 2 we will discuss the setup of the SEP focusing on the question of doing a real project. Section 3 outlines the domain and purpose of the tool APE to be developed and some technical background as well as the initial project plan. Section 4 describes what actually occurred, while section 5 summarizes lessons learned – not only by the students but by us as well. A short conclusion is given in section 6.

# 2: Project setup – Doing a Real Project

## 2.1: Students

The announcement of the course looked similar to a job advertisement. Besides offering real project experience, which is almost unique among university courses, the announcement emphasized the higher work load and time demand in comparison to other courses.

For successful team work during the course it is important that the differences between students with respect to motivation and knowledge are not too great. Therefore, as in real industry life we conducted interviews with all applying students. The major criterion for choosing a student was his/her motivation and commitment for doing project work. In these interviews 12 students from advanced semesters were chosen. This number of participants has proved sufficient for experiencing team work and the difficulties of communication.

There were even more factors that one can usually find in industrial projects: the team was brought together only for this project, and most of the members did not know each other before. Large parts of the used technology were new to most of the team members. The students were expected to spend a large amount of time on this project, but they had other courses on their schedules as well.

## 2.2: Customer

To provide a real setting for the project a “real” customer was involved in the course. A nearby company took over the role of the customer and provided an initial and slightly fuzzy requirement specification document for the software to be developed. Of course, this

requirements document was not a formal specification at all, but provided a lot of space for interpretation and clearly needed clarification. A further difficulty was the bulk of requirements given by the initial requirements document. Having too many requirements and only little time is not an untypical project situation. Students had to negotiate the functionality that can actually be done in a three months project with the customer.

During the project the customer was intended to be the primary address for the students' questions regarding requirements. After one presentation of the results in the middle of the project the customer was involved in acceptance testing. After the final presentation the customer had to decide whether to accept or to decline the delivered software. The delivery deadline was a hard deadline, i.e. the end of the term.

### **2.3: Teachers' Roles**

There were five supervisors (four staff members and one senior student). While four of the supervisors focused their work guiding their respective student team through the development process, the remaining supervisor was designated for quality assurance, thus being responsible for reviewing all the documents produced along the way. The intention behind this division of responsibilities was to assure his independence from the students and their work and his objectivity in reviewing these documents. Of course all other supervisors read and reviewed the documents as well.

Coaching included solving technical problems as well as moderating discussions about the use cases to be realized, the architecture of the application or planning of the next steps. Last but not least coaches had to maintain the motivation of the teams. The project leader (one of the supervisors) additionally took care of further organizational issues and kept in contact with the customer.

### **2.4: Limitations as Compared to a Real Project**

This course tried to be as close as possible to reality. But still, some differences remained. The main goal of teaching sometimes conflicts with the aim of a real-life setting. There were three months time – just enough to go through a full life cycle of a small software project. But there was no chance to touch maintenance issues.

The project started with 12 students, and each had to be occupied from day one. Typically, team sizes vary over time with some developers deciding on the architecture and a larger team joining in when it comes to implementation. In every real software project there is at least one experienced architect. In our project we wanted all students to do the entire job, including software architecture. We had inexperienced students and therefore could not expect them to design a perfect software architecture.

This approach makes sense for teaching, but it becomes a real problem if you want to get a solid architecture. But the teaching goal was to let the students experience the challenges of designing a good architecture. We believe that we did achieve this goal.

The incentives for developers, however, are quite different. In this project, no real money changed hands. Instead, students received credit points for attending. The product was meant only as a prototype and not for productive use, which seemed to change the attitude quite remarkably. The customer was not a “real” customer who was anxious to use the tool but a company willing to support this kind of university project.

### 3: Background for the Project “APE”

**The Living Software Development Process** Today’s software development projects are confronted with a frequently changing environment: rapidly altering business domains and processes, fast technology evolution, great variety of evolving methods and development processes. Therefore, a highly flexible and adaptable software development process is required that allows a project to react to changes quickly and to adopt existing development methods to comply with the project’s actual needs. We call such a process a living software development process – a process which allows tailoring according to a given project situation as well as evolutionary process improvement. The concept is based on process patterns. Further reading on the concept of process patterns can be found in [2], [5], [6].

**APE** The tool APE (“Applied Patterns Environment”) was to be developed in the project. Project managers can use APE for configuring a project plan based on their company’s process pattern library. APE supports the tailoring a generic process model containing alternative methods for the specific project. During project execution the team members are guided through the project by means of process patterns. APE can suggest applicable process patterns based on the state of the documents under development. For example, the pattern “find test cases” might be considered useful when the use case document reaches the status “accepted”.

Although APE remains a prototype it shows that the concept of process patterns is sufficiently powerful to provide a basic platform for a living software development process, which guarantees the required balance between flexibility and control in a process model. For more information on APE see [7].

**Technical Background** Practical SE requires know-how of a broad range of technologies upon which the project is based, and tools which are used in the development process. This project was no exception, quite a few tools and technologies were new to the students.

CVS (concurrent versions system) was used for version control. LaTeX was standard for writing all the documentation. TogetherJ [12] was used as UML editor for modeling and code generation; a number of Java libraries were used: JUnit [10] for regression testing and a setup for daily build and smoke tests, XML libraries (we used JAXP [11]), and Log4J [1] which provides a flexible and powerful logging mechanism. The Bugzilla bugtracking System [9] was installed and used extensively.

Finally the Eclipse IDE [4] was development tool and target platform at the same time: APE was designed as a plug-in for Eclipse, which meant extending and using the Eclipse APIs, like the SWT library (standard widget toolkit) for the GUI. Eclipse was the greatest technical challenge because it is a very young technology. Not many documentation resources existed describing how to extend Eclipse or use the services provided by its APIs.

Altogether there was an impressively broad range of tools and technologies with which not all students were familiar. Surprisingly most students said this was not the primary challenge for them – being either an indication for their high motivation and preparedness for learning new technologies, or indicating that other problems arising in the software development process are much more challenging.

**Team Organization** We divided our 12 developers into three teams of four students each and assigned them according to the layers of a classical three-layered architecture. Negotiation among the teams was necessary in several areas. During requirements clarification each team was responsible for writing a different part of a requirements specification document. In the end this document, of course, had to be consistent and complete. The teams were responsible for designing and negotiating interfaces between the layers. Since little experience regarding interfaces was available, a change board comprised of representatives of each team had to discuss necessary changes of the architecture document.

Coordination among teams was achieved through weekly project meetings attended by all participants. During the meeting each team reported accomplished work and further steps were discussed. The meeting results were captured by protocols, which were written by revolving students authors. Besides this weekly meeting, the teams contacted their supervisors on a regular basis where team-specific problems were discussed.

#### 4: The Actual Project Run

The available time of three months was divided into five phases<sup>1</sup>, as listed in table 1. At the time of project kick-off, the team agreed to this plan. It was considered to be ambitious but feasible. Minor deviations in the plan were expected, but there was no serious doubt about the success of the project. In the project run it turned out that some tasks were much more time-consuming than initially thought. Table 1 gives an overview of the temporal deviations.

Project Phase	Planned Duration	Actual Duration
Analysis phase resulting in a requirements specification document	3 weeks	5 weeks
Design phase leading to a architecture document, containing the detailed descriptions of interfaces between the layered architecture	2 weeks	4 weeks
Development of the first increment including integration, testing and delivery	4 weeks	6 weeks
Development of the second increment	4 weeks	—
Integration as well as acceptance testing	2.5 weeks	0.5 weeks

**Table 1. The Initial Plan and the Actual Project Run**

In retrospect the initial plan was overly optimistic (which might also be considered typical for software projects). The delays of the analysis and architecture phases led to the necessary plan adjustment of formally dropping the second development increment. We had to inform our customer that some of the use cases promised only a few weeks ago in the requirements specification might not be realistic for the remaining project time.

Not even this adjustment allowed the continuation of the project with the planned functionality and with proper documentation. In contrast to the official plan, the team (including the project leader) decided to achieve the initially planned functionality. In the

<sup>1</sup>Some of the mentioned phases did overlap to fit into the project duration of three months

end, about three days remained for the integration of the separately developed layers. For acceptance testing by the customer we had one day left.

During the implementation phase we experienced necessary changes to the interface specification document almost daily, although the document had been formally reviewed by the whole team.

**Quality of the Developed Software** In the end, the resulting software is not too bad. In spite of some issues regarding user guidance, the customer was satisfied with the result. Among the features realized there is full access to the imported XML data produced by a neighbour system, and the tool integrates nicely into the Eclipse IDE. On the other hand, APE falls short of supporting concurrent access by different users, but this deficiency was realized early and accepted by the customer through his signing of the requirements specification.

Finally, we were able to deliver a productive version of APE with limited functionality and a prototype version realizing almost the entire functionality as planned albeit with minor deficiencies. The code contains 10.500 lines of code (LOC, blank lines and comments excluded) in 120 Java classes.

## 5: Lessons learned

**Communication** Students were asked to fill in an anonymous questionnaire about the course with questions about the achievement of teaching goals and the experiences they gained. They all agree that the most difficult issue in software development is not a technical one, but a social one.

What is the right way to communicate in a larger group? There are about 700 emails in APE's mailing list archive (25 of them sent after 10pm). In this course, we did not answer the question of the perfect way, but students learned about the importance and difficulty of communication.

They also learned how to communicate with the customer. It was not possible to deliver all the functionality the customer ordered – and the students had to explain why some of the ideas could not be part of the first roll-out. There were good reasons, but they had to be presented in an appropriate way.

**Technical Issues** Students learned a lot about practical software development: They had to manage several tools as well as some new technology, and were required to handle an amount of code with continuously increasing size. Only towards the end, where the project became too complex for a single programmer, they realized the importance of modularization and stable interfaces by experiencing an overly increasing need of communication.

**Process Knowledge** Students also learned about the software development process in two ways: The domain of their project was about software processes, and they experienced one full development cycle.

“... Conducting a project from the beginning until the end was very interesting, as one saw the major differences between large projects and just a few days of programming on your own. Also seeing the supervisor's tasks (even if

you are just the one being supervised) was interesting. . . . “ (from a student’s questionnaire)

**Management Problems** Not only the students learned in this course – it was also a very interesting project from the supervisors’ perspective and a good way to experience practical problems of project management. The project plan had to be adjusted several times. There was serious misjudgment about how time consuming certain activities can be. What are the reasons for that?

First, the way from the initial customer specification to our requirements specification was much longer than anticipated. Knowing the domain, we initially thought it might be a small task. Second, before the start of the project we did not invest much time in a thorough evaluation of our base technology Eclipse. The GUI widgets of Eclipse turned out to be a rather difficult technology, especially when you have to use “trial and error” because of insufficient documentation.

Another mistake was that the list of features to be realized was not adapted according to the actual state of the project. Although the development phase for the second increment was dropped, due to our unfounded optimism, we tried to catch up with the original plan. At the end of the project we did not manage to stop coding at the proper time, in order to be able to test and properly document the software: The major part of code was written and tested in the last 2.5 weeks before delivery.

**Documentation** The elaboration of the requirements specification as well as the architecture document took much time, and was of poor quality in the end. Why was that?

“...Subject of future improvement might be the communication of tasks. Sometimes it was not clear at all what the expectations were – causing some delay of the project . . . “ (from a student’s questionnaire)

The early process phases were only roughly planned and prepared. We used a document template for the requirements specification, which turned out to be difficult to use, because explanations for its application were missing. For the architecture document, we only provided oral instructions, but no template, exemplary document, guidelines, or checklist. It is no surprise that the architecture was not a stable document but kept being changed continuously throughout the following phases. Also some useful architectural issues are missing completely in our document, for instance the description of the exception handling concept. This lead to problems during implementation and some deficiencies in the resulting software.

All in all the documentation does not seem to facilitate the further development of the tool APE – some reverse engineering might be necessary. On the other hand, the code itself is thoroughly documented using JavaDOC comments. We even have a comment ratio of 37 percent compared to the total number of lines of code.

Clearly, in providing useful templates together with checklists, we see great potential for improving the project performance.

**Team Organization** Another problem arose from inflexibility regarding the size of our development team. Untypically for a real project, we did analysis and architecture with the full team size of 12 developers.

From the teacher's perspective the approach has been valuable. All participants were involved in all life-cycle phases. In the end students may have learned the advantages of stable interface definitions. In the beginning of implementation, the GUI team gave the impression of not being able to cope with the whole functionality because of difficulties with the Eclipse GUI widgets. At the same time, the application layer team did not seem to have much to do. This had some influence on the motivation of the whole team. At the end of the project, when Eclipse finally turned out to be manageable, we simply did not have enough time for both implementing application layer features and testing them properly. We considered rebuilding the teams as a reaction to shifted work load. This might have been advantageous, but in our opinion changing teams could also have resulted in a major time loss for the integration of new team members.

### **5.1: Why did it work anyway?**

Sometimes work was accomplished in a practical manner, clearly resulting in some trade-offs, as for example an imperfect architecture. But finally the software functioned – even after a dramatically shorter implementation and testing phase. Analysis and architecture activities did not lead to high quality documents, but at the begin of implementation developers seemed to have understood the application domain quite well. So the thorough preparation of the implementation phase led to much quicker coding.

During implementation some practices proved very helpful. We established a nightly build and test run and sent emails to all project members in case of failure. Developers committed new code to the CVS repository very carefully, making team work smooth.

We established a formal procedure for change management regarding changes to the interface specifications. A change management team had to be informed about necessary architectural changes, approve or reject them and distribute approved changes by email to all parties involved. Furthermore the use of a bug tracking system as a means of communication proved quite helpful. 234 bugs or reminders have been noted in APE's bugtracking system.

Integration and testing were quite short but nevertheless did work. During the implementation a large part of the requirements were tested using automated unit tests. Judging the quality and completeness of the test cases written, one must say, quality is rather poor, but even the focus on test cases seems to have led to careful development so that integration in the end did not produce major problems.

The most important factor that saved the project's success certainly was good communication. Although we did not have a perfect plan all the time, we always knew the actual state and problems of the project. A good working atmosphere, high motivation of all participants, sometimes resulting in late night work, helped us in solving (almost) all problems during the project. An important factor for the success was probably that all participants had fun.

### **5.2: Lessons that Still Need to be Learned**

Several things could be done better. Some open questions remain.

**How can we get closer to reality?** The SEP still has kind of a sandbox character. To make the project more realistic we should not give the same type of tasks to all the

students. Furthermore, in a real project team size would vary over time. On the other hand three months time is too short for changing teams or tasks frequently.

**How can we better reach our teaching goals?** It is hard to teach how important it is to write good documentation (and how to do it). The value of good documentation only surfaces during the maintenance phase, which was not included in the project time.

Asked about their estimation for the ease of maintenance or developing additional features in the future, students told us at the end of the project that they did not see any problems in understanding their code after a year's time. This is fundamentally in contrast with the supervisors' opinion. From the teaching point of view we were not able to provide the "maintenance experience".

## 6: Conclusion

Is this the right way to teach SE? We believe it is, even though we still see shortcomings in setting up a real industrial project environment. Every year we are supported by companies that invest time and money just for playing the role of the customer. We interpret this fact as a sign that our approach helps provide students with the knowledge and experience that industry expects from computer science graduates.

By no means would we argue against studying standard SE text books. Courses like the one presented here demonstrate that we need both: Studying SE methods and creating an atmosphere as close as possible to real software project life.

**Acknowledgements** We are grateful to Rainer Frömmling for acting as the customer and Robert Abright for his thoughtful comments on a draft version of this paper.

## References

- [1] Apache Software Foundation. *Log4j*, 2002. <http://jakarta.apache.org/log4j/>.
- [2] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A Componentware Development Methodology based on Process Patterns. *Proceedings of the 5th Annual Conference on the Pattern Languages of Programs*, 1998.
- [3] Bernd Brügge and Allen Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice-Hall, Inc., 2001.
- [4] The Eclipse Project. *Eclipse*, 2002. <http://www.eclipse.com>.
- [5] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. Towards a Living Software Development Process Based on Process Patterns. *Lecture Notes in Computer Science*, 2077, 2001.
- [6] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. Modular Process Patterns Supporting an Evolutionary Software Development Process. *Lecture Notes in Computer Science*, 2188, 2001.
- [7] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. Towards a Tool Support for a Living Software Development Process. *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.
- [8] Steve C. McConnell. *Software Project Survival Guide*. Microsoft press, 1997.
- [9] Mozilla.org. *The Bugzilla Bugtracking System*, 2002. <http://bugzilla.mozilla.org/>.
- [10] Object Mentor, Inc. *JUnit*, 2002. <http://www.junit.org/>.
- [11] Sun Microsystems, Inc. *Java API for XML Processing (JAXP)*, 2002. <http://java.sun.com/xml/jaxp/index.html>.
- [12] TogetherSoft. *TogetherJ - The Model, Build, Deploy Platform*, 2002. <http://www.togethersoft.com/>.