A Practical Software Engineering Course with Distributed Teams

Egon Teiniker and Sybille Paar
Department for Internet Technology and Software Design
FH JOANNEUM
Kapfenberg, Austria
egon.teiniker@fh-joanneum.at

Regina Lind
ZML – Innovative Learning Scenarios
FH JOANNEUM
Graz, Austria
regina.lind@fh-joanneum.at

Abstract—Traditional didactical methods with a strong teacher-centred approach have proved to be inadequate for software engineering education as they do not capture the complexity of the software production process. Our paper demonstrates that software engineering can and should be taught in an authentic learning environment that relies heavily on a student-centred teaching approach. We present our experiences with an example of a practical software engineering course that simulates a real-world software engineering project carried out in distributed teams. This concept has been tried out and improved based on continuous feedback from students and professionals over the past four years. The proposed mixture of different pedagogical concepts (constructivism, experiential and collaborative learning), that also reflect certain agile software development practices, has significantly improved the quality of software engineering education.

Keywords—e-learning; experiential learning; collaborative learning, agile software development

I. INTRODUCTION

Teaching software engineering in a classroom is a challenge because important software engineering techniques cannot be put into practice when adhering to traditional didactical approaches: Conventional student projects tend to be too small, they start with specifications that are too clearly defined, and they are implemented by student teams which are not big enough to mirror real-world software engineering projects. Software, however, does not happen by accident. It requires fundamental engineering knowledge in requirements engineering, software architecture and design as well as skills in coding, testing, configuration management and software maintenance. For this reason software engineering needs to be taught in an environment that closely resembles the situation of a real-world software project.

This paper therefore introduces a didactical concept of a software engineering course that emphasizes the importance of student-centred, collaborative learning in an authentic environment. The main purpose of this paper is to show that the quality of software engineering education can be improved if certain aspects of already existing pedagogical concepts are combined in a way that the learning activities the students engage with reflect the methods used in agile software development.

The paper is structured as follows: Section II introduces the theoretical background of agile methods in software engineering and the didactical concepts our course is based on. Section III gives a detailed account of the organization and implementation of our practical software engineering course. Before the concluding chapter, section IV sums up the results by emphasizing the similarities between the pedagogical concepts used for this course and the methods applied in agile software engineering. Finally, the lessons learned part gives an overview of those aspects that should receive special attention when adopting a similar didactical concept.

II. RELATED WORK

A. Software Engineering Education

Software engineering is an engineering discipline that is concerned with all aspects of software production: It comprises the early stages of system specification as well as the maintainance of the system after it has gone into use [32].

Taking the complexity of engineering topics into consideration, the design of didactical concepts for engineering courses is challenging. So far, the following shortcomings in software engineering education have been observed [28]:

- As classroom projects tend to start from scratch, students are not able to read or re-engineer existing source code.
- Student projects which are implemented in small groups within a semester are too small to show the benefits of proper software engineering practices.
- Even though students learn most from the feedback they receive on their mistakes, essential feedback is very often missing as lecturers find it too time consuming to give extensive feedback on an array of student projects carried out each semester.
- Software quality is not taken into account as software projects of student teams are often evaluated and graded on their functional aspects only.

For the reasons mentioned above, some teaching approaches where courses are more focused on learning-by-doing rather than on lecture-based concepts have already been developed. In these learning environments students gain the ability to apply their knowledge and skills to real-world problems [4], [21]. An analysis on how different personality types respond to this
new learning environment is given by Layman [24] who found out that an agile approach to software engineering education addresses the needs of students with different learning styles.

Other courses are based on peer reviews where students review the work of their fellow students and thus get more feedback on their programming work. This works, according to Safran, even with a large amount of students in class [25].

B. Agile Methods in Software Development

Since the beginning of software engineering, more and more complex software process models have been defined and used. The emergence of Web-based software systems has posed new challenges to software engineers. Frequently changing requirements, short time to market and rapid technology improvements have raised the need for new software development process models: the so-called agile methods.

Agile methods are based on two concepts: the honesty of working code and the effectiveness of people working together with goodwill [20].

Currently, there are various types of agile methods used in the industry. The most well known of which are Scrum [27], [26] and Extreme Programming (XP) [5], [6]. While Scrum focuses on management aspects of software development, XP describes a set of related software development practices. Regardless of the differences between these implementation variants, the following concepts are common to all agile methodologies [13]:

- **People-orientation:** While classic process models see developers as replaceable parts, agile methods identify people as the most important factor in software development.
- **Iterative development:** Everything in software development depends on the requirements. Therefore, classic process models are focused on extensive requirements engineering and exhaustive requirements specifications. The problem is that the requirements will change even in Web-based applications over the project duration.

Agile methods divide a project into time equivalent iterations. Each iteration involves a requirements analysis, a design, coding and testing stage. At the end of each iteration a running software product with a subset of functionality can be released to the customer. After each iteration the customer can change, add, or remove any requirements which are documented in so-called User Stories. A user story is a short description of a valuable feature of the software system that fits on an index card.

- **Self-adaptive process:** In agile methods, the process model itself is reviewed after each iteration. In a retrospective session the developers analyse what worked and what did not in the last iteration. These reflections serve as the basis for continuous adaption to generate a process model that works well for the whole team.

The scope of agile method application is usually restricted to teams of four to ten members. For larger projects several teams have to work in parallel. Another major aspect of agile methods is communication. In an ideal programming environment all team members have the possibility to sit together in an open-plan office that enables face-to-face communication. In projects that require many and probably even distributed teams, collaborative online tools can narrow the existing communication gap [35], [18].

C. Pedagogical Concepts

As has been propagated by many educators, software engineering education needs to become more realistic and learner centred [16]. To overcome the already described shortcomings in software engineering education we have based our teaching approach on different pedagogical concepts and combined them in a way that is intended to enhance the students' learning experiences, facilitate the learning process, increase the students' motivation and thus improve their skills in software engineering techniques.

**Experiential learning** is one of the concepts used for our teaching approach as it “...seeks to place what is learned in a context that is real, in the present, and shared among others. This promotes better integration of knowledge and skill, better retention, and better transfer of tasks” [7]. Experience, according to the experiential learning theory, is considered to be “raw material to be acted upon by the mind through the controlled and self-conscious use of the senses (observation) and the application of reason (reflection)” [34].

Kolb [22], one of the main advocates of experiential learning, claims that experiential learning takes a “holistic integrative perspective on learning that combines experience, perception, cognition and behaviour”. Learning, or the formation of knowledge, can thus be seen as an integrated process that starts with the learners experience in the here-and-now. Any observations gained from this experience are then analysed and reflected upon by the learner. The conclusions and insights drawn from this reflective process are turned into a theory that is finally tested in new situations. Learning, according to Kolb, is hence considered to be an iterative cycle that is divided into four stages as visualised in Fig. 1.

![Figure 1. Experiential learning cycle: “The learner tests new concepts and modifies them as a result of reflection and conceptualization activities” [22].](image-url)
Concrete experiences in the here-and-now are the basis for learning to take place. In order to improve the effectiveness of the learning process and to guide students through the learning stages, thus avoiding that they either get bogged down in the analysis and reflection stage or move on to the testing stage too quickly, feedback is of paramount importance [22]. Through appropriate and timely feedback any deviations from the intended goals are assessed and the students’ actions redirected.

The lecturer thus turns from the producer or the source of knowledge into the facilitator who supports the learners when support (feedback) is needed, guides the learning process when guidance is necessary and creates additional possibilities for further experience [34].

The aspect of creating authentic learning environments propagated by the experiential method is also an integral part of Constructivism, the second approach which has had a significant impact on our course design in software engineering. According to Oliver and Herrington [19] constructivist principles “…posit that learning is achieved by the active construction of knowledge supported by various perspectives within meaningful contexts. In constructivist theories, social interactions are seen to play a critical role in the processes of learning and cognition.”

Learning, in other words, needs to be embedded in realistic and relevant contexts that are meaningful to the students and similar to those environments in which students expect to apply the acquired knowledge [4]. If students are confronted with real-world problems, they experience an authentic learning situation which enhances their knowledge construction [17], [15]. Furthermore, when students actively participate in the knowledge construction process they develop a sense of ownership and responsibility for their ideas and their thinking, as their views are heard and accepted by others in the respective collaborative learning situations [10]. For this reason, constructivism propagates an approach towards learning where students are no longer the passive recipients of knowledge transmission but take an active part in their own knowledge construction and become responsible for their own learning [7].

This shift from teacher-centred to student-centred learning, which is an integral part of constructivist learning, is also central to the educational approach of Collaborative Learning. In collaborative learning activities students work in pairs or groups in order to find solutions to a given problem, establish meaning or create a product on their own. As students bring many diverse experiences, backgrounds and multiple perspectives to these learning activities, an “intellectual synergy of many minds” [31] needs to be established. This involves a process of mutual exploration, meaning-making and feedback which does not only support and enhance the students’ understanding of a topic, but which can also lead to the creation of new knowledge [31].

A similar idea is expressed by Cunningham et al [10] who argue that in constructivist learning environments, learners need to also experience the constructedness of knowledge, which means “that any ‘truth’ begins with a set of untested assumptions which can be examined to evaluate the adequacy of the position taken”. These multiple perspectives, which form the basis for a negotiation of meaning, can only be provided if learning takes place in a social context. If students have to construct knowledge collaboratively by engaging in discussions, dialogues and information sharing activities with other students, learning will take place on a far higher level than by merely listening to a lecture on the same topic. Thus, meta-cognitive and higher-order thinking skills, like for instance critical thinking, logical reasoning, self-evaluation, reflection etc., are trained and developed in collaborative and constructivist learning environments [17], [15].

III. A PRACTICAL SOFTWARE ENGINEERING COURSE

We have developed an undergraduate software engineering course which is based on the pedagogical concepts described and simulates a real-world development environment for our students. In this practical software engineering course, students - usually a group of 20 to 30 - work together on a single project.

A. Organization

Over the last four years we have implemented and improved this concept with two different student groups per year: One group consists of full-time students and the other one of part-time students who are taught in a blended learning environment where 60% of the classes are taught online via an open source eLearning platform and 40% are held face-to-face at the university. The students are grouped into teams of 5 to 8 members and the lecturer takes over the roles of the customer and the technical consultant.

1) Learning target: The aim of this practical software engineering course is to practice the software engineering knowledge of students in a simulation of a real-world project. Students learn how to use their theoretical knowledge about software engineering and how to improve their collaborative working skills.

Our practical software engineering course focuses on techniques which are commonly used in industrial practice:

- **Agile development methods**: Agile software development is very common in the domain of Web applications because the requirements of Web-based software systems are changing frequently. We use Scrum as a management framework and implement some development practices from XP, such as on-site customer, simple design, collective ownership, testing, refactoring, continuous integration, and short releases.

- **Software architecture and design**: We focus on the functional and non-functional aspects of Web-based applications. Software architectures and designs [14], [12] are a major topic in this course because design is an
important activity to improve the quality, modularity, and extensibility of software systems.
- **Continuous integration**: To work efficiently within a team and also correctly between teams, all artifacts created by students have to be managed. This implies source code versioning, automatic build, -tests and -deployment of applications after each change of the existing code base [11].

2) **Educational requirements**: Before students enter this practical software engineering course in the 5th semester they have passed several lectures in the context of software engineering:

- **Software engineering basics**: Overview about the main activities in software development, and the basic concepts of common software process models [32].
- **Data structures and algorithms**: Analysis and implementation of traditional data structures and algorithms [9].
- **Object-oriented programming**: Introduction to the concept of object-oriented programming. We use Java as a language to practice these concepts because Java and many related tools and libraries are available for free and Java is also easy to learn [3].
- **Software and database design**: Based on object-oriented basics, students learn about design- and architecture patterns and how to use and implement the right pattern for a given design problem [14], [12]. Students learn the design of database schemas and the interaction between the relational and the object-oriented paradigm [2].
- **Software quality engineering**: Software quality can be improved through testing and code analysis. Both techniques together with methods to design efficient test cases are part of this course. Students learn that not only the functional but also non-functional aspects of a software system must be specified and validated [36], [23], [37].
- **Web programming**: The implementation of Web applications is complex. Students have to deal with concurrency, session management, scalability and security. Simple examples are used to show students how these techniques work [1].

At the beginning of the 5th semester, students have learned most of the theoretical basics of software engineering and have gained experiences when working on smaller projects in teams of up to three people. Now they are ready for the simulation of a real-world project developed in distributed teams.

3) **Team structure**: To manage the work of so many people, a class of 20 to 30 students is divided into four teams. Two types of teams and two roles for the lecturer are defined as shown in Fig. 2:

- **Customer**: The lecturer plays the role of a customer who presents his visions and ideas for a new software system to the students. The customer is responsible for setting priorities for the application’s functionalities and gives feedback at the end of each iteration.
- **Technical consultant**: The lecturer plays also the role of the consultant who advises students on technical problems and methodical discussions. Usually, a consultant is needed when problems in the project appear.
- **Development team**: Each of the three development teams has to implement functional and non-functional requirements ranging from the user interface to the database. As a development team has to deal with all layers of a Web application it is called a cross-functional team. For this reason, each development team needs to be able to implement a complete end-to-end feature of the software system. From a formal point of view, a development team consists of five different roles: team manager, requirements engineer, software designer, software programmer, and software tester.
- **Integration team**: As in larger software projects certain roles and activities are common to all teams, a separate integration team was established to take over these roles and activities in order to avoid any redundancies. An integration team consists of the following roles: project manager, configuration manager, software architect, quality manager, and documentation writer.

4) **Development and communication platform**: As a platform for development as well as for the communication between developers we used Google Code [8] which is available

---

**Figure 2. Roles in the practical software engineering course**: The lecturer plays the role of the customer and the technical consultant. The students are grouped into three development teams and one integration team.
for free and supports a range of useful services for students and lecturers:

- **Project homepage**: Students can create a website as an entry point to their project.
- **Source code repository**: All project artefacts are stored in the repository. Thus, each student has access to all documents and source codes. A history of all changes, which is useful for monitoring student activities, is recorded automatically.
- **Wiki pages**: Wiki pages are used by students to create tutorials, to write documentation or to post some notes about a technical problem.
- **Issues**: Issues are structured text messages with a unique identifier, so that students can use them for user stories and bug reports.
- **Download page**: Additional documents and product releases can be downloaded from this page.

5) **Grading**: The evaluation of the individual performance of students who work together in teams is a challenge: Some students tend to work more than others and they also have different levels of expertise and background knowledge in the respective fields. Furthermore, their skills develop at different rates. Therefore, the grading scheme of this course is based on the evaluation of practical as well as theoretical tasks and exercises:

- The outcomes of the practical work of each iteration is rated per team. Thus, all members of a team get the same ratings.
- In a final class exercise which asks for details on the whole project, students are individually evaluated.

This evaluation mix ensures that students engage in both the teamwork activities and the engineering parts.

**B. Implementation**

The whole semester is divided into three iterations and a leading startup phase. According to the agile principle of short releases, a running version of the Web application is delivered after each iteration.

1) **Startup – Preliminary Work**: Before the first iteration starts, students get an introduction to the course organization, the principles of Scrum and the Google Code platform. A critical part in this setup phase is the team building process. Existing student groups are mixed in order to ensure balanced teams of experienced and novice developers.

Within this phase students learn how to use the Google Code platform and particularly the source code repository. The team members have time to establish their internal communication structures and to become a working team. This phase usually lasts three weeks.

2) **First iteration – Make it work**: At the beginning of the first iteration, the customer presents his/her visions and ideas to the teams. This communication between customer and teams is verbal and informal. The teams can ask for details and create user interface prototypes on the blackboard.

The customer’s ideas and wishes are transformed into user stories which are stored as Issues in the Google Code platform. All user stories from all teams are collected and displayed on the platform and the customer is able to set priorities. Based on the customer’s priorities and the team’s estimations an iteration plan can be created. This plan describes which user stories can be implemented by which team in the first iteration.

Then, each development team is responsible for implementing the planned user stories which are usually end-to-end features containing user interfaces, business logic and database access.

The integration team is responsible for setting up the project structure, the automatic build process and the manual system tests.

At the end of the first iteration a running Web application is presented to the customer. Each student writes an experience report that reflects what went well and what did not work out the way it was planned. Suggestions for improvements are also made. In a retrospective session, improvements for the second iteration are discussed.

All three iterations have the same duration of four weeks each. It is a principle of agile software development that iterations are of equal length.

3) **Second iteration – Make it better**: The focus in the first iteration is on fast implementation of functional requirements and students tend to produce quick and dirty code without a sound design and test cases. This code comes close to real-world legacy code. During the second iteration the focus shifts to software quality, redesign and refactoring.

The second iteration starts with extensive code reviews. Poor design and code smells are identified in class room review sessions. Manual reviewing is supported by static analysis tools to improve efficiency. Reviews have a positive impact on the knowledge transfer between students.

Test cases are designed from the implementation structure and implemented in test suits. Refactorings must be performed to make the existing code testable.

All refactoring and test implementation requests from all teams are stored as Issues on the Google Code platform. They are estimated and prioritized and used for the second iteration’s plan.

The development teams perform refactorings and implement automatic test cases. Usually, an overall software architecture is defined and design patterns are applied.

The integration team adds the automatic tests and analysis tools to the build process and configures a continuous integration server which monitors the source code repository and triggers a new build after repository changes.

At the end of the second iteration the running Web application is presented to the customer to show that the required software changes have been implemented successfully. Additionally, a continuous integration cycle illustrates the automation of the build and test process. A final retrospective session highlights possible improvements for the next iteration.
4) Third Iteration – Make it right: During the second iteration, the customer has changed his/her mind and presents new ideas to the teams, e.g. mobile clients are more important than any other missing feature. Following agile methods, we can re-prioritize user stories and plan the third iteration.

Based on the knowledge of iteration one and two, the development teams are able to finish the implementation during the third iteration. They can learn from their mistakes and benefit from the progress they have made. Therefore, a feasible product can be presented to the customer.

The integration team automates the system tests and writes a user documentation for the Google Code platform’s Wiki system.

During a live demo session the final release is presented to the customer and all team members. Also the project’s homepage and Wiki will be reviewed. In a final retrospective the last iteration as well as the whole project will be reflected and discussed.

IV. RESULTS

As mentioned before, we have operated and improved this course concept over the last four years. We have changed the organization and the implementation aspects in many directions. The following results should give an overview of the concepts that have been considered as useful.

A. Pedagogical Concepts and Agile Methods

Our teaching approach has been influenced by different pedagogical concepts. Surprisingly, we found many similarities between these pedagogical concepts and agile software development methods.

From the perspective of Experiential and Constructivist Learning the following analogies can be identified:

- Students obtain a real-world problem, namely to realize a customer’s vision of a Web-based software system.
- A real-world project is simulated by applying the same methodologies and tools that are used in the industrial practice.
- When working on their projects, students learn by doing. They use their theoretical knowledge to solve practical problems.
- The lecturer plays the roles of the customer and the technical consultant and is no longer a pure knowledge transmitter.
- In the code review sessions, technical problems are observed and reflected. This reflective process which is accompanied by the lecturer’s feedback supports the students in their knowledge construction.
- In the retrospective sessions taking place after each iteration, organisational and communicative processes as well as the students’ experiences gained in the respective iterations are analyzed and discussed.
- The review and retrospective sessions help students form abstract concepts and generalized approaches that are tested in the next iteration of the agile process model.

At the same time we can find the following similarities to Collaborative Learning:

- Students work together in distributed team and all teams work on the same project.
- Reviews and retrospectives are applied group work activities where different perspectives and knowledge backgrounds are discussed to find a good solution.
- Many different ways of communication are used in this project among students and between the students and the lecturer.
  - Direct communication: Face-to-face discussions and group work activities in the classroom and the labs.
  - Synchronous online communication: For direct communication between distributed teams, students use the Internet, e.g. Skype [30] or TeamSpeak [29] and Virtual Network Computing (VNC) [33]. TeamSpeak and VNC are integral parts of our blended learning platform. Thus, students are used to communicate this way.
  - Asynchronous online communication: As distributed teams do not always work at the same time, students use online services like email or discussion boards. A great tool set for asynchronous communication is provided by the Google Code platform. Issues and Wiki pages are used intensively during the whole semester.

The observation that agile software development methods use well known pedagogical concepts does not only open up new perspectives for software engineering teaching practices but also for knowledge management and life-long learning activities within software companies.

B. Lessons Learned from Practical Software Engineering

The following lessons have been learned while carrying out the practical study outlined above:

- For students it is very important to experience what it is like to deal with a customer who does not have a clear idea about the product’s features. Thus, the role of the customer played by the lecturer can help students to better understand the problems related to requirements engineering and specification.
- The role of the consultant is very effective as it offers the lecturer the possibility to discuss and solve any problems the students might come across exactly where and when they appear. Students tend to be very interested in learning about techniques that help them overcome problems they currently have.
- Special consideration should be given to the team formation as students should have the opportunity to learn from each other. For this reason, students with varying levels of experience and expertise in software engineering are put together in the same group.
- Setting the focus on software functionality in the first and on software quality in the second iteration brings all the bad development practices to light and helps to improve the students’ skills in agile software development.
• A retrospective review after each iteration (even after the last iteration) is extremely valuable for the students’ learning process. Students feel that they can change and improve things.

• Using approved methodologies and tools from the industry and the open source community allows a realistic project simulation in an academic context where students can test their problem solving competences without commercial risks.

As a side-effect of this course our department has learned a lot about the requirements for a well designed software engineering curriculum. Over the last four years we have identified and closed various knowledge gaps with new or improved courses in the lower semesters.

V. CONCLUSION

We presented our concept and the experiences derived from our practical software engineering course in which we performed a semester project with 20 to 30 students organized in four teams.

Based on agile software development methods and the Google Code platform we simulated a real-world development scenario with distributed teams. By altering the simulation conditions like focusing on functional aspects in the first and quality aspects in the second iteration as well as changing the requirements, we forced students to apply their software engineering knowledge to solve realistic problems.

From our experience we can say that an iterative approach with retrospectives, explicit customer and consultant roles for the lecturer as well as extensive reviews can improve the quality of software engineering education significantly.

We also identified several similarities between experiential learning, constructivism, and collaborative learning concepts and the agile software development practices.

REFERENCES