Teaching the Software Engineering Process Emphasizing Testing, Rationale and Inspection (TRAIN)

Lars Borner
Institute for Computer Science
University of Heidelberg
Im Neuenheimer Feld 326
D-69120 Heidelberg, Germany
borner@informatik.uni-heidelberg.de

Barbara Paech
Institute for Computer Science
University of Heidelberg
Im Neuenheimer Feld 326
D-69120 Heidelberg, Germany
paech@informatik.uni-heidelberg.de

ABSTRACT

Students learning software systems development at the University of Heidelberg follow the TRAIN process. This tool supported process emphasizes the early phases of software development by combining different activities and artefacts such as requirements engineering, quality assurance as well as graphical user interface and system design. By the use of TRAIN students are being enabled to finish large software projects. TRAIN focuses especially on various decisions (rationales) in the development process and on several quality assurance activities like testing or inspection. This paper gives a short overview of the TRAIN–process, how it is taught in software engineering courses and how it is supported by a tool called Sysiphus.

Keywords

Teaching, software engineering process, tool supported education, requirements engineering, test, inspection, rationale

INTRODUCTION

At the University of Heidelberg software engineering skills are taught and applied in various software engineering courses. Students learn basic skills in the course “Software Engineering I” and in the practical courses for beginners. They acquire more profound skills in advanced courses like “Software Engineering IIa” (focusing on requirements engineering and project management), “Software Engineering IIb” (focusing on architectures based on component technologies and web services) and practical courses for advanced students. These courses are designed especially for bachelor and master students in computer science, but students of other fields of study are invited as well.

In all software engineering courses students have to handle a software project of realistic dimension in a given time. The intention is to allow students to gain experiences in developing or extending large software systems. In these projects one focus lies on early phases of the development process like requirement elicitation (~30%) and system design (~25%). The other focus is quality assurance (30%). We assume that programming skills are already available. All projects are performed in teams. This allows students to refine their soft skills [8] in team work and communication.

The development is based on the well defined TRAIN-process. The CASE-tool Sysiphus [12] supports TRAIN by assisting students in performing different activities and documenting required artefacts.

The reminder of this paper is organized as follows: Section 2 shortly introduces TRAIN and its main activities. The third section describes the various artefacts of TRAIN in detail and how activities and artefacts are taught in the course “Software Engineering I”. The last section summarizes our experiences with TRAIN and Sysiphus in our courses and discusses the advantages and disadvantages of our approach.

TRAIN – THE OVERALL APPROACH

The TRAIN-process combines different concepts of software engineering, emphasizing **T**esting, **RA**tionale and **IN**spection to support the early development phases. We have developed our own approach with detailed guidance for requirements engineering covering also GUI design and non-functional requirements (NFR), because the requirements engineering approaches we found in the literature do not integrate all these aspects. Therefore we adapted the Rational Unified Process (RUP) by adding easily understandable methods of requirement engineering and system design supporting rationale and quality assurance activities. In the requirements phase the TORE approach [10] is used to identify and specify the relevant functional and non-functional requirements. This is complemented by the approach of [7] to design user interfaces and to create first prototypes of a GUI. The approach proposed in [3] is used to transfer requirements into a class design.

Similarly, we could not find detailed guidance for the development of test cases in parallel to the requirements. Thus, we incorporated our own advices into TRAIN on how to derive test cases for the system, integration and unit test, and give hints on how to specify these test cases. To make sure that all specified artefacts in documents are

correct and consistent, TRAIN includes various types of inspections like perspective-based or checklist-based inspections.

An important but often forgotten part of software documents is the documentation of decisions made during the development process. Typically these decisions are implicit only in the results of these decisions. However, the discarded options and the decision criteria are lost. Therefore, often during changes and extensions (e.g. through personnel not involved in the development) important criteria are forgotten, discarded options are re-discussed and consequently the decision quality is reduced. Thus, TRAIN demands to document this rationale (see [2]); i.e. the different options, reasons and decisions that have been involved.

In the following we shortly describe the basic ideas of the approaches within the early phases in TRAIN. Afterwards we sketch the CASE tool Sysiphus and explain how it supports TRAIN.

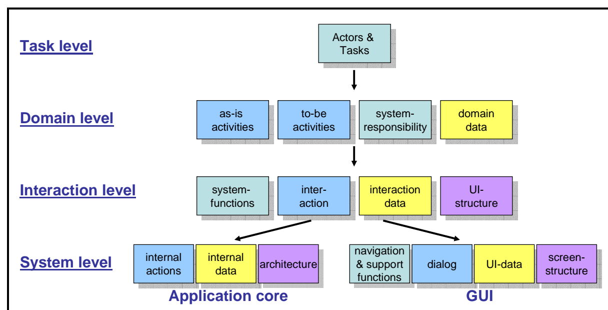

Figure 1: Levels and decisions of TORE

TORE

TORE (Task Oriented Requirement Engineering) was first introduced in [10]. It describes a "conceptual model for the decision types [and NFRs] that should be supported by methods integrating RE and OO" (see [10], p. 49). These decisions are arranged in four abstraction levels: task level, domain level, interaction level and system level. These levels and decisions are illustrated in Figure 1. At the top level roles and tasks of different business processes are identified. At the domain level the requirement engineer analyzes how these tasks are currently performed (as-is) and how they should be performed in the future (to-be). Using this information, the requirements engineer can define activities that have to be supported by the new software system (system responsibility) and the required domain data. At the interaction level the interactions between users and the new system are specified by defining system functions, use cases, interaction data and user interface (UI)-structures. The lowest level is divided into two parts. One deals with the application core and the other concentrates on the graphical user interface.

GUI-approach

In [7] the author describes an approach to derive the Graphical User Interface (GUI) design and to develop the first GUI-prototypes. In a first step the tasks and data of the domain level are analyzed to identify several workspaces. A workspace contains all activities and data of one or more tasks and represents a first abstract description of parts of the later GUI. An example of such a workspace is illustrated in Figure 2. It shows the workspace of the "Select Books" task for an online book store.

In a second step the workspaces are refined into views (virtual windows) by using information that is contained in use cases. In the last step the navigation and the support functions are added to these views to get dialog and screen structures. These structures can be realised by several kinds of prototypes such as mock-ups or functional prototypes.

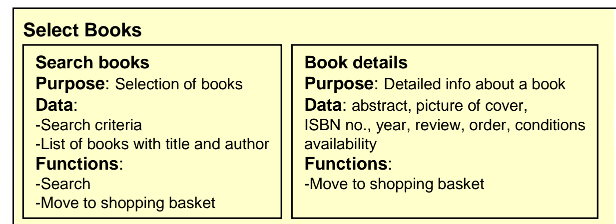

Figure 2: Example of a workspace description for a "select books"-task

Design-approach

Jacobson describes in [3] an approach to develop class models. He uses artefacts specified in the requirements phase especially artefacts in domain data diagrams and use case and system function descriptions. First, analysis class diagrams are created by using entity, boundary and control classes. Then boundary and control classes are transformed into "real" classes, attributes or operations, and basic and complex operations are identified and specified. Finally, the distribution of attributes and operations to classes is revised, based on sequence diagrams for the system functions. The goal is to achieve loose coupling and high cohesion between classes.

Rationale-approach

In [2] Paech and Dutoit introduce an approach for documenting rationale. They use the question-option-criteria (QOC) approach to make decisions explicit. Made decisions or decisions to be made are documented as questions. Every question possesses one or more options, whereas an option describes one possible solution alternative to this question. Criteria are assigned to every question and are used to assess the proposed options. Having assessed all options for one question, the best option can be chosen as the best solution for this question and the decision is recorded in the corresponding specification document.

Inspection-approach

In TRAIN inspections are applied to uncover defects in the specification, i.e. to find inconsistent, incomplete, duplicated or ambiguous parts. The documents are read by one or more inspectors. For this purpose two different reading techniques can be used: the checklist-based or the perspective-based technique. The former uses a checklist to find defects. The inspector reads the document and tries to detect defects according to defect classes that are defined in a checklist. The latter uses different roles. Every role corresponds to a special point of view, e.g. a tester point of view. The inspector gets a task he/she has to solve according to this role. Usually it is easier to discover defects, when working actively with the document based on a task.

Test-approach

TRAIN supports test case design as well as test case execution. System, integration and unit test cases are derived based on the specification. Use cases and information of the GUI prototypes are used to design test cases at the system test level. We are applying the testing techniques "equivalence partitioning" and "boundary value analysis" (see [1]) to derive test data for system test cases. At the integration test level sequence diagrams and state charts are used to develop test cases. Class diagrams are the basis for test cases at the unit test level. All these test cases have to be designed before the next level of the software development process can be reached, i.e. all system test cases have to be specified before the design phase can be processed. After the programming phase all test cases are executed to uncover bugs.

Sysiphus

Sysiphus [12] is a CASE tool developed at the Technische Universität München (TUM). Its special purpose is to support teachers and students to teach and learn software engineering skills. With different kinds of documents Sysiphus provides a well structured support for TRAIN. Sysiphus allows specifying requirements artefacts in the Requirement & Specification Document, design artefacts in the Object Design Document and test artefacts in the Test Specification Document. Furthermore, it allows the documentation of rationale by asking and answering questions during the development phases. Every question is assigned to at least one element specified in one of the documents mentioned above.

By supporting distributed cooperative work, Sysiphus facilitates on the one hand inspections of one or more documents by more than one inspector at a time and on the other hand team work of two or more students. One possibility to access the documents contained in Sysiphus is to use the web-front-end REQuest. A screenshot of REQuest is shown in Figure 3. The left side of the figure shows the structure of the Requirement & Specification document. Here you can see different elements that can be documented in Sysiphus, e.g. actors, user task, use cases or system functions. The right side of the figure shows

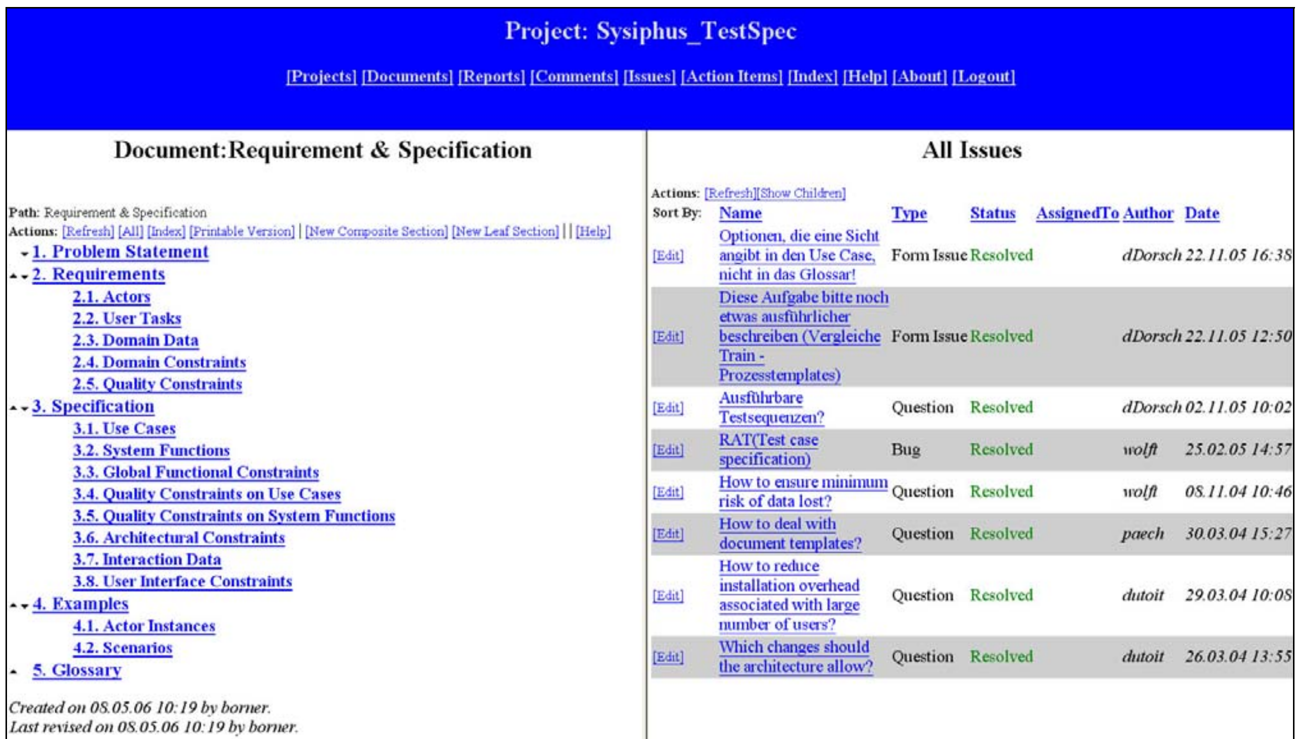

Figure 3: Screenshot of the REQuest-GUI of Sysiphus

the documented rationale for the current project represented in question form. Every question can have two different states: open and closed. All questions in Figure 3 are closed, i.e. for every question the best option was chosen and the decision was documented. The second possibility to access documents is to use the Swing-GUI RAT. While using RAT the user does not need an additional program like a web-browser to access all documents in Sysiphus. All he/she needs is a Java virtual machine.

Sysiphus provides templates for nearly all artefacts of TRAIN. Furthermore it allows to link artefacts together e.g. actors to user tasks, test cases to use cases, questions to system functions or classes to packages. This allows easy navigation between documented elements on the one hand and traceability between artefacts of different development phases on the other hand.

TEACHING TRAIN

In the course "Software Engineering I" students become familiar with TRAIN. We divide the semester in two parts. In the first seven weeks we teach basic knowledge and practice this new knowledge in small exercises. In the last six weeks students have to finish a software project of realistic dimension by applying the newly gained skills. The task is to extend an existing software system with new functionalities.

Knowledge and Skills

The necessary knowledge is taught in a weekly 2-hour lecture course. Every week we introduce new activities and artefacts of TRAIN. To illustrate these activities and artefacts small examples are used. In the first half of the semester the new knowledge and models are practiced in small, weekly and independent exercises. Here three different approaches are used. First, students have to complete an incomplete model, second, they have to find and correct inconsistencies between models and third, they have to correct a given incorrect model. These exercises refer to the existing software system that has to be extended. The advantage is that students become familiar with this system. All artefacts and activities are practiced with examples of this system i.e. the students have to "re-engineer" several artefacts of the given system. In addition the students learn how they have to document the artefacts in Sysiphus. To motivate the students for this approach, the tasks are embedded into small scenarios like this: students are new employees of a large software company and their first task is to restore lost models and artefacts of a given system.

We introduce the artefacts and activities bottom up, that means we start with the design and component test and move on to more abstract descriptions in requirements and integration and system test.

After a short introduction into TRAIN and Sysiphus, students get to know the different models and artefacts of the design phase. For example they have to complete a given class diagram by using the source code. They learn how the different model elements of a class diagram are mapped to the source code and vice versa. They also reconstruct sequence diagrams out of the given source code and the control flow of the system.

At the same time students get to know how to derive test cases for unit tests from class and sequence diagrams and source code. They become acquainted with different coverage criteria like statement, branch or path coverage and with different testing techniques like equivalence partitioning and boundary value analysis. They design and document these test cases within Sysiphus and realise and execute the test cases by using JUnit [5], e.g. students had to develop test cases for a given class. In this exercise they have to reach 100% branch coverage to successfully finish the tests for the given class.

In the next weeks we practice the TORE levels in the requirement phases and teach all required activities and decisions. We start at the interaction and system level. At these levels it is important to teach different kinds of NFRs. The students have to become aware of the importance of NFRs. For example they have to redo some decisions in the existing system. They have to decide which architecture the system should have. Therefore they have to evaluate different given options against a set of given NFRs and choose one option that supports the NFRs best. For this purpose we created a question in Sysiphus in the summer term 2004. We added all possible options and identified all important NFRs. First, the task of the students was to assess every single option against given NFRs in a matrix. Afterwards they had to choose the best solution and compare this solution with the current architecture of the software system. Figure 4 illustrates the structure of such an assessing matrix. The different options are documented in the upper left corner. In the right half of the figure the criteria and assessments can be seen. As you can see the option "3-layers installed on one machine" fulfils the given criteria best.

At the interaction level the students also get to know new artefacts like use cases and system functions. Moreover, they learn to write their first use cases, to design use case diagrams and to specify complex system functions in Sysiphus. They become acquainted with the right abstraction level of use cases and system functions und become able to fill in the templates given in Sysiphus. They identify the pre and post conditions as well as several steps of use cases. At the same time they are taught to create their first abstract description of a GUI: the workspaces and UI-structures.

After specifying use cases and system functions, the students begin with their second quality assurance activities. On the one hand, they develop integration test cases for given system functions. They identify the classes that realise the system functions, analyse the interactions between these classes regarding the system functions and document integration test cases in Sysiphus.

Question: *What is the best architecture for the new system?. (borner, 08.05.06 10:35)*
Assigned:
References: Architecture
Decision: *Question not resolved yet.*

Criteria [Revise Assessments] [Select Criteria]

Options [Propose Option]	Easy extendability for new functionality	Global availability	No new hardware	No new software	Runnable on MacOS, Linux and Windows	Secure connection to documents	Small amount of data
[Edit][Delete] *3-layers installed on one machine. (borner, 08.05.06 10:34)*	--	--	++	+	--	++	++
[Edit][Delete] *3-layers realised within a Client/Server architecture. (borner, 08.05.06 10:32)*	-	+	NA	NA	+	-	NA
[Edit][Delete] *4-layers realised within a Three-Client/Server architecture. (borner, 08.05.06 10:31)*	+	++	--	-	+	-	-
[Edit][Delete] *Repository pattern realised within a Client/Server architecture. (borner, 08.05.06 10:33)*	++	+	-	-	NA	-	-

Figure 4: Rationale example

Afterwards they use JUnit to realise and execute these test cases. On the other hand, they develop scenarios, in order to describe exemplary instances of the use cases. These scenarios are used to specify the first system test cases. Here Sysiphus also provides templates to describe scenarios and test cases. Afterwards every scenario and every system test case is linked to a use case. To realise and execute the system test cases, the students have to use JWebUnit [6].

At the task and domain level the students learn how to identify roles, tasks and domain data. They practice specifying actors and tasks by restoring lost actor and task descriptions. They link together existing actors, user tasks and use cases with restored ones in Sysiphus. Furthermore, they practice modeling domain data in an entity relationship diagram and specifying NFRs for the domain level. The NFRs have to be identified and specified correctly in Sysiphus and linked to corresponding user tasks by the students.

So, in the first half of the semester the students have practiced all important development activities, but not in a coherent method. So in the second half of the semester – simultaneous to the extension task – the students learn how these artefacts are developed coherently top down. In particular, they learn to derive the class model based on the specified use case and domain data model. They become familiar with boundary, entity and control class and how these elements are used to transform the domain data diagram into analysis class diagrams. Furthermore they are taught to validate the class model by using sequence diagrams.

Project work

Students apply the new knowledge and skills they acquired during the first seven weeks to extend an existing system. The system they have to extend is the same system they had used to specify and document their artefacts: Sysiphus. In this phase, Sysiphus is no longer a CASE tool to the students. It is also the software system they have to extend. Sysiphus comprises more than 700 classes and over 100.000 lines of code. In our opinion it is a software system of a realistic dimension. The main advantage in using Sysiphus as the software system to extend is that students already know how to use the system and how it is realised. The most important parts of the system are familiar to them by now. To give the best possible support, they get access to all existing documented artefacts, diagrams and models of Sysiphus including test cases and the source code. Furthermore, they get a handbook of TRAIN. This handbook can be found in [11]. It contains an overall description of the process illustrated by an example. It can be used as a reference book, where they can look up all details of TRAIN.

In teams of four or five, the students have to perform the extension task. As an example we would like to mention the project of the summer term 2004. The students had to add new inspection functionality into Sysiphus. They analysed the inspection process and identified the activities that could possibly be supported by the tool. For example they had to distinguish between two different kinds of defects an inspection process can uncover: semantic and syntactic defects. They had to understand that only the syntactic defects can be found automatically.

One example for such a syntactic defect could be a missing initiating actor for a user task. In Sysiphus every user task has to have an initiating actor, otherwise the user task is not complete.

Every project has a strict time schedule and three given mile stones. Every two weeks one of the mile stones has to be reached. Moreover every team gets its own tutor. The teams have to send partial results to their tutors every week. The tutors give detailed feedback on these results. To reach a mile stone the teams have to present their results to a fictitious customer, mostly the professor of the course.

One focus of the project work lies on functional and non-functional requirements. The students have to create descriptions of actors, user tasks, use cases as well as system functions and of course they have to document decisions the team members have made. Figure 5 shows a diagram similar to a use case diagram that represents the solution for the extension task of summer term 2004. The teams had to identify and to describe the role of the inspector. For the inspector they specified the user task "inspect requirements" (the dark grey oval in the diagram). This task was realized by two different use cases ("Inspect requirements manually" and "Inspect requirements automatically"). The latter was supported by a system function "Execute automated inspection" that describes how the later system performs the automated inspection. The non-functional requirements have to be documented for the requirements phases as well as for the architecture and design phase. One example of an NFR of the project of 2004 was the demand that an automated inspection had to be executed within one minute. In addition to the functional and non-functional requirements dialog, screen models and the user interface prototype play an important role and have to be developed within the project. In 2004 every team developed at least one prototype for the later extension of Sysiphus. Therefore they used different kinds of tools. One team used a paint tool and another team realised the prototype with HTML. The prototype was added to the documented elements in Sysiphus.

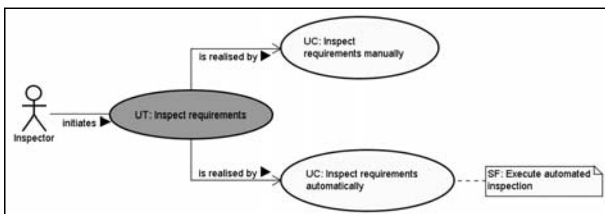

Figure 5: Use case solution of the extension task

The second focus lies on quality assurance activities and artefacts. On the one hand the teams have to specify and execute system, integration and unit test cases. Before a milestone can be reached, the test cases of the corresponding development phase have to be specified, and after the implementation the test cases have to be executed. On the other hand every team has to inspect the results of another team and give detailed information about uncovered defects. In a session of 90 minutes the teams have to uncover as much defects as possible. Here we use different kinds of inspections. At the requirements mile stone the students apply the perspective based technique. Therefore four different perspectives are used: tester, designer, customer and rationale maintainer perspective. The checklist based inspection technique is applied before the design mile stone is passed. After every inspection every team gets the possibility to correct its defects.

The third main focus of the project work lies on design models. The teams have to derive the class model stepwise. They are developing analysis class diagrams, class diagrams and sequence diagrams. In summer term 2004 students had to create about 10 new classes, adopt about 15 classes and interact with nearly 50 classes. The number of lines of code differed from team to team and was between 2.000 and 8.000 lines of code.

Similar to the exercises in the first half of the semester the teams have to document all artefacts they are creating in Sysiphus. To draw different kinds of diagrams like use case, class or sequence diagrams, we use the UML-tool Jude [4]. Diagrams drawn with Jude can be exported to JPEGs and attached to the corresponding documents in Sysiphus.

EXPERIENCES

In the previous sections we have introduced an approach to teach activities, artefacts and models of early development phases. So far our approach was used in the courses "Software Engineering I" in the summer terms of 2004 and 2005. A possible curriculum for the courses can be found in [9]. In 2004 ten and in 2005 twenty five students attended the courses. At the end of the semester the students had to answer questions about the course. E.g. questions like: "Did you enjoy the course?" "How important was the team work?" or "How much did you learn about the software engineering process?" Most of the answers were very positive. All students were convinced that they learned a lot in the course and most (~ 80 %) of them enjoyed our course.

These last two semesters have shown that detailed feedback to the students is one of the main factors of success. The feedback should be given not only in the second half of the semester but also in the first half. Teachers should point out the mistakes the students have made immediately. Subsequently the students should have the time to correct their mistakes. To show that solutions of one software engineering task can be quite diverse, the different solution proposals have to be discussed in weekly exercises. Most of the students agreed in the questionnaire that the immediate feedback was very important for the learning success. But some of them annotated that the feedback could have been more detailed.

The main advantage of our approach is that the students learn the basic knowledge and skills in the first half of the semester. This knowledge is taught in weekly lectures and skills are practiced in small, mostly independent, weekly exercises. In order to really embrace the knowledge and skills, the students have to finish a project of realistic complexity from beginning to end. The project "Extending Sysiphus" is large enough to confront the students with real problems of software projects. In the point of view of most of the students (~ 70%) the extension task within Sysiphus was very important and assisted to understand the main phases of a software development process.

Team work is the second advantage of the approach. It reduces the extent of feedback needed. It is easier to give feedback to four or five students at once then giving feedback to every single person in particular. Furthermore the students learn to work in teams, to communicate with customers and to organise themselves. Nearly 70% of our students agreed that team work was very fundamental for them. They learnt to communicate with and to coordinate each other and to give hints and assistance to other team members. For three students only the communication overhead was too large and they argued the team work was hindering.

Supporting TRAIN with Sysiphus helps to reduce mistakes beginners often make by providing templates to document required artefacts at different development phases. Furthermore the tool alleviates the communication within the teams, because all team members can work on the same project documents and discuss different possible solutions at the same time. The third important benefit of Sysiphus is that it enables teachers to give feedback to all team members easily by (mis-)using the rationale functionality. The tutors are able to question confusing or inconsistent parts of the student's documentation. He/she creates a new question in Sysiphus and describes the misunderstandings and links the question to the corresponding elements. The students can describe how they solve the problem as an alternative option and close the question. They also can use the rationale functionality in Sysiphus to discuss different solution with the tutors.

The main disadvantage of our approach is the very high amount of support given by teachers and tutors. On the one hand giving detailed feedback takes a lot of time. Every week the solutions of the students have to be commented and discussed within every single team. For the next semester we plan to assign the task of giving feedback to students of a higher semester. This could disburden other tutors and the given feedback would be more detailed.

Sysiphus is enhanced continuously. That means existing examples of artefacts used in the first part of the semester have to be updated before the next semester starts. But enhancement of Sysiphus is done by the tutors of the course. So they are familiar with the different parts of the software system and the corresponding source code. This helps to give the best possible support to students during the extension task.

Another weak point of TRAIN is that Sysiphus does not support all activities of TRAIN in the same way. There could be better support for deriving the design model from the use case model. Of course one can document different artefacts like use cases, classes as well as packages and link them together. But so far it is not possible to document the results of intermediate steps like classes of the analysis class model.

The analysis of the questionnaire has also shown that the amount and the complexity of some tasks seem to be too large for the students. Most of them spent more than nine hours a week to solve the tasks. Some of them argued, that within the semester the extension task of Sysiphus is too time consuming and could be better realised within the holidays in a three week compact course. Maybe this is a possibility to reduce the work load of the students within the semester.

Nevertheless, at the end of the questionnaire most of the students pointed out that they would recommend the "Software Engineering I" course to fellow students. We are convinced that the combination of TRAIN and Sysiphus supports teaching very well. So far, our approach was used in teaching small groups of students only. However, we are sure it could be used with larger groups of students provided there is enough capacity of teachers and tutors.

ACKNOWLEDGEMENT

We would like to thank Anke Borner, Timea Illes, Andrea Herrmann, Doris Keidel-Müller and Dima Suliman for their helpful reviews.

REFERENCES

1. Binder, R. V. (2003) Testing Object-Oriented Systems – Models, Patterns, and Tools, Addison-Wesley
2. Dutoit, A. and Paech, B. (2002) Rationale Management in Software Engineering, Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing Company
3. Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. (1998) Object oriented software engineering – A use case driven approach, Addison-Wesley
4. Jude (2006) http://objectclub.esm.co.jp/Jude/
5. JUnit (2006) www.junit.org
6. JWebUnit (2006) http://jwebunit.sourceforge.net/
7. Lauesen, S. (2005) User Interface Design – A Software Engineering Perspective, Addison-Wesley
8. Lichter, H., Melchisedech, R., Scholz, O. and Wiler, T. (2003) Erfahrung mit einem Workshop-Seminar

im Software Engineering-Unterricht, *Proceedings of SEUH 8: Software Engineering im Unterricht der Hochschulen, Workshop des German Chapter of the ACM und der Gesellschaft für Informatik e.V. (GI)* Berlin, dpunkt Verlag, 89 – 100

9. Paech, B., Borner, L., Rückert, J., Dutoit, A.H. and Wolf, T. (2005) Vom Code zu den Anforderungen und wieder zurück: Software Engineering in sechs Semesterstunden, *Proceedings of SEUH 9: Software Engineering im Unterricht der Hochschulen, Workshop des German Chapter of the ACM und der Gesellschaft für Informatik e.V. (GI)* Aachen, dpunkt Verlag, 56 – 67
10. Paech, B. and Kohler, K. (2003) Task-driven Requirements in object-oriented development, In: Leite, J. and Doorn, J. (2003) Perspectives on Requirements Engineering, Kluwer Academic Publishers
11. Häfele, P. (2005) Softwareentwicklung mit dem TRAIN-Prozess - Bachelor Thesis http://www-swe.informatik.uni-heidelberg.de/research/publications/reports.htm
12. Sysiphus (2006) http://wwwbruegge.in.tum.de/Sysiphus