Int. J. Engng Ed. Vol. 24, No. 4, pp. 738–746, 2008
Printed in Great Britain.

0949-149X/91 $3.00+0.00

Adopting Lakatos in a Software Engineering Course*

NICHOLAOS PETALIDIS
Department of Informatics and Communications, Technological and Educational Institute of Serres, Serres, Greece. E-mail: n.petalidis@computer.org

The standard practice in a software engineering course is to present the theory as a list of dogmatic guidelines. In this setting problems appear artificial and consequently students fail to appreciate them. Similarly, solutions arrive magically, letting students believe that this is the norm. The value of an incremental and iterative methodology is therefore missed. A different approach, borrowed from Lakatos [1], is presented here. Students are given a problem and through 'trial-and-error' discover their own solutions. Unlike a typical term-project that follows the theory, it is the problem that drives the theory. The result is better appreciation and comprehension of software engineering notions.

Keywords: software engineering education; constructivism; reflective learning and teaching

INTRODUCTION

IN HIS SEMINAL WORK, Lakatos [1] presents a dialogue between a teacher and his students. The dialogue starts with a problem: whether there is a relation between the number of vertices, *V*, the number of edges, *E,* and the number of faces, *F,* of regular polyhedra. The students after many attempts, notice that

$$V - E + F = 2 \text{ (Euler's formula).}$$

The discussion carries on and progressively the students rework both the formula and the notion of a regular polyhedron.

The work is usually considered as an attack on meta-mathematics and the 'formalist' school. But it is much more than that. Throughout the text Lakatos also comments on the way mathematics is taught. For example on p. 142 [1], Lakatos notes:

> Euclidean methodology has developed a certain obligatory style of presentation. I shall refer to this as 'deductivist' style. This style starts with a painstakingly stated list of axioms, lemmas and/or definitions. The axioms and definitions frequently look artificial and mystifyingly complicated. One is never told how these complications arose.

Lakatos' view of mathematics and his method of proofs and refutations are quite similar to the way that software is developed: someone builds a solution, and a series of tests. If the tests fail, the solution is reworked and new tests are devised and so on until the various notions involved in the problem become clear and a solution is found that doesn't fail the tests. This correspondence between Lakatos' 'proofs and refutations' technique and software engineering is not of course a new observation. Researchers have drawn upon it before, for example in [2] and [3].

However, Lakatos' constructivist approach to teaching has only recently started being considered in software engineering courses.

In this paper the results of such an approach are presented. The paper proposes a 'proofs and refutation' iteration, after which the notions highlighted through the first iteration are taught formally with traditional lectures. The proposed methodology can be classified as a problem-based methodology of teaching because the students learn by working on an open-ended problem with an unknown solution. It can also be classified as a collaborative-learning methodology, as students are encouraged to work with other students in informal group activities.

The paper is arranged as follows. First a review of other approaches to constructivist learning and their relation to the one presented here is made. Then the standard practice in teaching software engineering is presented together with an analysis of problems inherent to teaching software engineering. Following this, the proposed course structure based on Lakatos is described together with a working example. The paper concludes with the presentations of the conclusions from the experiment.

RELATED WORK

There are various works that describe attempts to deviate from the traditional lecturing approach to software engineering education.

In [4] the design of a two-year undergraduate software engineering program with emphasis on exploratory learning is presented. The program considers lectures to be supporting activities;

* Accepted 7 June 2005.

knowledge is mainly built by hands-on exercises and a set of project courses where students actively search for solutions to the problems they experience. The teacher plays the role of an advisor that helps the student understand how the content taught fits into the greater picture.

A similar plan is reported in [5] where the design of Carnegie Mellon's master of software engineering curriculum is presented. Students have the opportunity to practice in a studio under the supervision of a coach, much like students in architecture practice in a studio under the supervision of a teacher. In the studio, students work on a project that spans many semesters. The coach is responsible not for supplying solutions but for providing suggestions, asking why-questions and helping students maintain the vision of the project. In such a setting, students learn through reflection. The goal of the studio-approach is to provide a place to practise techniques learned in the core courses. Similar approaches, where intense student–supervisor communication in a studio environment replaces the traditional lecture-hall interaction, are also presented in [6] and [7].

The above proposals are basically problem-based, collaborative learning approaches that require students to get actively involved in learning activities and they can be broadly classified as constructivist educational methodologies [8]. In this respect, they are similar to the one presented here. They however describe a comprehensive plan that requires restructuring of the whole curriculum; this might not always be possible. They also represent activities that run in parallel with lecturing, in contrast with the flow presented here.

Another suggestion for a software engineering course based on active learning is presented in [9]. There, student learning is based on a team project that is accompanied by mini-lectures with immediate in-class applications, peer-teaching and small group discussions. Students also practise their writing skills and critical abilities through guided writing exercises, Toulmin analysis of articles and minute papers. The benefits of group-discussions are also reported in this paper, but the rest of the course structure proposed here differs.

Perhaps the methodology that most closely resembles the one presented here can be found in [10], where all lecturing from the course was removed and replaced by interactive class discussions and small group lab-work where students asked questions and added the missing pieces in their understanding of the project. In here however, lecturing is not removed but used in a subsequent iteration to emphasise and formalise the acquired knowledge.

Apart from experimentation with different course structures, researchers have also tried different technologies and techniques to deliver active-learning courses. One such approach can be found in [11] where the Web is employed to assist an active learning methodology and in [12] where the use of an educational simulation game is described.

THE TRADITIONAL COURSE STRUCTURE

Software engineering is a subject that is taught in most of the computer science curricula around the world. Its main purpose is to provide students with a background in executing large software projects and offer a glimpse of the processes followed by software companies. It is generally considered a difficult subject to teach as other authors have also mentioned (e.g. [13, 14]) and the results are, frequently, less than ideal: the software industry has complained several times that the level of education of future software engineers is not satisfactory [15, 16, 17].

In most cases the course is taught through a series of lectures that go through the topics of software methodologies, requirement capture, design and implementation, testing and maybe even project management. It is also common, in an effort to help students experience the practical side of software engineering, to combine the lectures with a small term-project that, too, follows the same sequence.

Popular textbooks on the subject include (but are not limited to) [18, 19, 20]. These textbooks usually present their material in a sequence. For example in [18] the material is developed according to well-defined phases of software engineering: project management, requirements capture, design, testing, etc. Even the proposed term-project follows this sequence.

But why are there so many complaints? This lecture-based approach, accompanied by some lab work or short project is the same approach that is followed in other courses, for example image processing or computer networks.

We believe that the main problem is the complexity inherent in the course. Software engineering is about executing large projects sensitive to requirements about quality, redundancy, maintenance and reusability. The knowledge required to carry out such projects comes from different backgrounds such as computer science, or management and psychology. As [15] notes, 'software engineering is a multi-faceted discipline'. In addition software engineers have to operate in a rapidly changing environment, since theories, methodologies and tools may soon become obsolete [4].

Thus, when someone is designing a software engineering course he or she has to take into account all of these problems and this is not easy.

For a start, as several authors have noted (e.g. [12, 14, 15, 21, 22]) it is impossible to reconstruct projects of the required complexity in a classroom. Instead the students either experiment with toy-projects or are guided with so much attention that they never experience the problems that software engineering is supposed to solve. In these cases the students are presented with a series of the 'proper' steps they have to follow in order to produce a quality software product: write down requirements, make an analysis, design and implement

according to standard programming guidelines. The problems that may arise if they don't follow the rules are of course briefly mentioned. But most students fail to grasp the extent of the damage that may be caused to the project in this case. After all, most of them up to this time have only worked on very small projects that came with explicit requirements and there was no need for software reuse and extensibility. (A similar remark is also made in [23].) As a result, they cannot relate to the concepts taught. The same problem is also reported in other studies [13, 14].

For example, students are taught programming guidelines and the text presents what should go into the comments section of the code. Students will duly note that, and they will probably write some comments in this style should the lecturer specifically requests it next time they hand in an assignment. Most of them consider it a burden or something that is 'nice to have' but 'not absolutely necessary'. Most of them will also complain if they are marked down for failing to put proper comments. This is something that should be expected, though, because the current educational practice fails to pass on to them the importance of proper commenting in a large setting. As [14] notes: 'Which teacher has never experienced the greatest difficulties to convince students that comments are essential in a program? The documentation of the source code is, in their eyes, useless.'

Similarly, in the course, students may be taught about cohesion and coupling. Students will write down the various coupling and cohesion levels, note the 'bad' ones and mark the advantages of the 'good' ones. However the student never realises how really 'bad' a program with high coupling is. It is common for students that scored high marks in written exams to have at the same time submitted code for their term projects that contained lengthy functions (over 100 lines for a single method) with multiple nested levels of if-else and switch statements. This shows that even though the students have learned all the right things they still find that it is unnecessary and time-consuming to actually apply what they have learned. And one of the problems that they consistently mention is that they feel that it is not necessary to go to the extra length of proper design, as long as their project executes as it should.

Even the notion of iterative and incremental development usually fails to make an impression to the students. After all, the lectures follow the waterfall-like approach present in the textbooks. The students will hear about the advantages of an iterative methodology but the reality is that they will not experience it in the lectures. Some times they might not even have the time to experience it in the lab [24].

Thus the inability to reconstruct a real-life setting prohibits the students from understanding and appreciating the ideas present in software engineering [13].

In addition, students will probably fail to see the multi-faceted nature of software engineering and will more often than not play down the importance of non-technical skills required in a large project. As [25] notes about the classical approach to teaching software engineering: 'Learning targets that go beyond technical questions are hard to teach in this kind of setting.'

Moreover, in order to maintain currency it is important to teach students skills that will allow them to carry on learning on their own, after their graduation. But such self-directed learning is not a skill that can be acquired during the traditional lecture-based methodology where the students' learning solely depends on the lecturer's presence.

Interestingly enough though, prior students of Software Engineering modules, doing their placement year in rather disorganised small software companies, have reported that they realised the value of the software engineering course only after they saw the failures of ad-hoc development. It was these comments that provided the incentive to try a different teaching methodology.

AN ALTERNATIVE COURSE STRUCTURE

In order to overcome the difficulties mentioned in the previous section a different approach was tried out. The idea was to slowly instil in the students the basic pillars of a successful software project, through an example.

In this setting, the whole course was divided in two parts.

The purpose of the first part was to illustrate the process of software development through a working problem. The purpose of the second part was to revisit all the areas that were highlighted during the first part in a more formal setting. In software engineering terms there were two 'releases', the first being an investigative beta release and the second an official release.

It should be noted here that the working problem mentioned above does not serve the same purpose as the usual term-project that students undertake during a software engineering course. In the latter the project follows the theory. In the former it is the problem that drives the theory.

In the first part, each lecture starts with a problem. The students discuss the problem and propose solutions. The role of the teacher is, through the questions and counter-examples he or she makes to guide the discussion so that the roots of the problem and the constituent parts of each proposed solution are made clear. Problems do not come out of the blue. Each problem comes as a consequence of a previous problem and an assorted solution. Through this process the students learn how to ask the right questions and how to improve on their solutions. More importantly they also see how and why software engineering practices are developed. The second part

reiterates the lessons learned in the first part. During the second part the students learn through the traditional lecture-based approach and carry on developing the project that they started in the first part.

In both parts students are expected to work in a group. Such a collaborative model of groups between two and four persons has been frequently suggested in the literature (e.g. [10, 11]) and has been deemed as beneficial.

At the end of the course the students sit an examination that focuses on theoretical issues. The result of this exam constitutes 50% of their final grade in the course. The other 50% is the individual project grade that each student received.

The project grade for each student is calculated as follows: At the end of each release each project is marked according to its adherence to requirements, implemented functionality, design and provided documentation and each team member has an oral interview in order to determine his/her contribution to the project. Thus for each of the two releases, a different mark for each student is calculated. The average of the two marks is the student's final project grade.

The alternative course plan proposed here is described better in Table 1. Note that the course is divided into two major parts. The 'Activity' column describes the activities during each week. During the first part most of the activity is a discussion on the problems the students faced and the possible solutions. During the second part the activity is mainly a lecture or a demonstration of a tool. The 'Project Deliverables' column describes what the students are expected to deliver before the start of the next week. The last column

Table 1. The suggested course plan

Part	Week	Activity	Project deliverables	CC2001 suggested SE core topics
Trial-Error approach 1st Part	1	Introduction to the course Project handout Discussion on the project's requirements	Software (Code+ Executable), v. 0.1 (Throw away prototype)	Software Requirements and Specification
	2	Discussion on requirements capture Techniques for writing down requirements	Software (Code+ Executable), v. 0.2 (Evolutionary prototype) Requirements document, v. 0.1	Software Requirements and Specification
	3	Discussion on Programming Guidelines Discussion on API Documentation	Software (Code+Executable), v. 0.3 (Improvement on programming style) API documentation	Using APIs Software tools
	4	Discussion on cohesion, coupling Measures of a design's quality	Software (Code+Executable), v. 0.4 (Improvement on cohesion and coupling) Requirements document, v. 0.2	Software Design
	5	Discussion on design patterns Short discussion on UML	Software (Code+Executable), v. 0.5 (Application of Design Patterns) UML Diagrams	Software Design Software tools
	6	Discussion on software methodologies Iterative/Incremental methodology	Software (Code+ Executable), v. 1.0 Short paper on the methodology followed	Software processes
Lecture Based approach 2nd Part	7	Software Processes (Waterfall, XP, RUP, etc)	2nd Release plan	Software processes
	8	Requirements Capture (Functional, Non Functional Requirements, Use Cases, etc)	Requirements document, v. 1.1	Software Requirements and Specification
	9	UML (Diagrams, Tools)	UML Diagrams Design Document v. 1.1	Software Design Software tools
	10	Software Design Issues Coupling, Cohesion, Architectures	Design Document, v. 1.2 Software (Code+Executable), v. 1.1	Software Design
	11	Design Patterns	Software (Code+Executable), v. 1.2 Design Document v. 1.3	Software Design
	12	Course Review	Software (Code+Executable), v. 2.0	

describes the relevance of the taught material to the proposed Software Engineering course contents of the ACM/IEEE Computing Curriculum 2001. Note that the course plan does not include content relevant to software project management, software quality and testing. These are expected to be taught in a subsequent more advanced course regarding software engineering.

A WORKING EXAMPLE

The proposed example and methodology was adopted when teaching the software engineering module at the department of Informatics and Communication at the Technological and Educational Institute of Serres, Greece during the Spring term of 2006–2007. This is a 12-weeks module, offered during the 6th term of the degree programme of the Institute. By that term most students are familiar with the basics of procedural programming and have taken an introductory course on OO programming and database design but have no major project assignments. During their 7th term students also take an 'Advanced software engineering' module that focuses on software management and quality issues, but this module is not described here.

In the past software engineering was taught in a more traditional way, introducing development methodologies with some emphasis on RUP, requirements capture, analysis and design patterns, programming and documentation guidelines all of them in sequence. The main problems from this approach were presented and analysed in the previous sections.

In this experiment, however, the first half of the course, i.e. the first six weeks, was designed around a central example which served as the basis for discussion and experimentation. The selected example was that of a logging library, in the line of log4j [26]. The example was chosen on purpose to be something that most students were not familiar with as none of them had developed a library before or even used logging somewhere in their previous works. The students were allowed to work in teams of up to three people.

The second half of the course followed the traditional lecture-based approach and was used to re-emphasise the lessons learned during the first part. During the second part the students were asked to carry on with their team-project and pair the library with an application that provided a graphical interface and a parser that could read and display logging messages in a GUI environment. In the following we present the iterations carried out during the first and second part.

Iteration 1.1

The purpose of the first iteration was to introduce to the students the importance of proper requirements capture. Students were given the project description which was intentionally short and ambiguous:

> You are required to write a logging library with support for different logging levels and different output destinations, (e.g. a console or a database)

No reference on how this project will be marked was given.

In this respect, the project started in a manner similar to that of many real-life projects: with a vague description by the customer, no agreement on when this project will be considered 'done' and lots of other unanswered questions.

Students initially argued heavily on the vagueness of the project but soon started asking questions to clarify it. Questions though were not focused but ranged from issues like 'what is logging' to 'can we use the X compiler' or 'do we have to use ADO or ODBC to connect to a database?'

Students soon realised that no matter how much they debated on implementation issues they did not progress much on understanding the project. So the discussion focused on what sort of questions would bring more valuable information, and what would be the best approach to understanding the customer's wishes.

Concluding the discussion, the students were asked to explore further on the projects' requirements by consulting the functionality of other libraries and present their first version of the solution. The students delivered an implementation of their idea of the required solution. This in effect was a throw-away prototype, that was used later to clarify the project's requirements.

Iteration 1.2

Upon delivery of the first version the students had a clearer idea of the requirements, but still lacked comprehension in many areas. The presented solutions were crude and somewhat inconsistent as different groups understood the requirements in a different way.

For example some solutions would only output messages if the logging level was exactly the same as that of the message, others would 'hard-wire' the required output destination at the time of the creation of the logger. Others would 'prompt' the user for the desired destination every time a message needed to be output.

However, this time questions were better aimed: 'How do you want to use the library?', 'What is a logging level?', 'Will there be a possibility of simultaneous output to different destinations?'

Also, once the initial confusion over the requirements started clearing up, the question on how we agree that the project was completed was raised.

This in turn proved to be a starting point for a discussion on techniques and the reasons for writing down and agreeing on requirements. Students were asked to write down the prerequisites for the project in order next time to agree on how the final deliverable will be graded.

Iteration 1.3

The second version showed a better understanding of the project and delivered a working prototype with written requirements. For most students this should have been a 'completed' version, had it been a different programming module. But this module put more emphasis on delivering 'products' rather than programs (in the notion described in [27]). In order to highlight this difference each group was asked to adopt another group's library to demonstrate its usage.

Groups, however, implemented the library in different ways and adoption was difficult. For example some groups had logging levels starting from 0 to 5, others from 1 to 6. Others were expecting one parameter to each loggin function, others had more and no documentation was provided. Parameters were named unintelligibly and it was not obvious from a first read what their purpose was.

So, the resulted discussion centred on methods to overcome these problems through proper documentation and coding standards.

Typical questions that were discussed were the following: 'What sort of comments will be useful to another person trying to read the code?', 'What is the best way to name variables etc.?', 'What sort of hidden assumptions do we make about parameters and how shall we protect against them?'

For the next version teams were given the task of reworking their code according to the clarified requirements and of improving both coding standards and existing documentation. A reference to the documentation tool Doxygen was also made in order to use it in their code.

Iteration 1.4

The third solution produced code that was better suited to the requirements and adhered to coding standards. In addition some documentation was provided. However coupling was high and cohesion was small. Typical solutions included code that looked as follows:

```
void error(
 int output,
 string message
)
{
 if (output == 1) {
 // append to console
 } else if (output == 2) {
 // send to db
 }
}
```

At that point students were asked to add more output destinations, like BSD sockets and files, and to think how they would go on redesigning their solution. This led some students in writing different functions as follows:

```
void writeToConsole(string msg);
void writeToDb(string msg);
void writeToFile(string msg);
void writeToNet(string msg);
```

This in turn generated a discussion on why they considered the second solution better than their first.

A typical answer from most of the students was: 'I wrote it as an independent function in order to be able to re-use it should the case arise', which in turn raised the question: 'Will you only need to re-use it in the environment of a Logger?' which in turn gave a good lead to discuss coupling and cohesion.

At the same time students were asked to amend their list of requirements to include the possibility of different layout schemes for each output. They were also asked to produce another version with a better design that reduced coupling and increased cohesion.

Iteration 1.5

The fourth version saw the delivery of software that was better designed in terms of coupling and cohesion, even though it still lacked excellence in many cases. Most groups had identified different classes for loggers, layouts and appenders (see [26]) but they made poor use of object-oriented features. A typical solution contained a class of the following form:

```
class Writer {
writeToConsole(string msg);
writeToFile(string msg);
 . . .
};
```

However this solution provided a basis for another discussion:

Do we need to group all of the function in a single class? What do all these functions share in common? Can we explore inheritance? What advantages do we get if we apply it in the code?

Also, questions on why these classes were chosen, or whether we could spot these classes from the start were asked. The similarity between the requirement to have different output destinations and different layout schemes for each destination was also explored. This led to a discussion on common patterns in a design and an introduction to the idea behind design patterns in general was made.

Students were asked to revisit their code and improve their designs according to the outcomes of the previous discussion.

Iteration 1.6

The fifth version of the code was by far the better one as students had applied the strategy pattern they 'discovered' in order to provide for a solution with low coupling and high coherence. They also added extra functionality to the library by including other output destinations and layout schemes.

At this point the students were asked to review

the steps that brought them to this point. A discussion on development methodologies started as they realised the iterative and incremental nature of the process they followed. A debate on the merits of the approach was initiated and other approaches were also discussed. The advantages and disadvantages of having short repetitions were also brought up.

The students were asked to write a paper describing the steps and the methodology they followed.

At the end of the first release the students delivered a requirements document and a short paper on the iterative and incremental approach that they followed. They also submitted around 1200 lines of code, API documentation in HTML format and UML diagrams that were generated by the Doxygen documentation tool.

Iteration 2.1

The second part of the course used the traditional lecture-based approach accompanied by a small project that reiterated the concepts learned during the first part. For this second part students were also asked to develop a parser and a GUI displaying logging messages as an addition to the project they developed in the first part. The students were expected to carry on with their teams, but this time without much intervention from the lecturer.

The first lecture formalised the theory behind software methodologies, explained the notion of iterative and incremental development and described the waterfall approach, XP and UP together with their prospective advantages and disadvantages. The students were also asked to submit a release plan for the second part of their project.

Iteration 2.2

The second lecture focused on requirements capture using various techniques (e.g. use cases, decision tables, BNF) and formalised the notions of functional and non-functional requirements. The students were also asked to deliver a requirements document for their project.

Iteration 2.3

The third lecture presented UML in a more academic setting and let students experiment with the use of a UML drawing tool. For their project the students were asked to submit relevant diagrams presenting the domain model of their project and flow of control.

Iteration 2.4 and 2.5

The fourth and fifth lectures were concerned with design, presented different architectural approaches, revisited the coupling and cohesion metrics for OO programs and described some of the GoF design patterns. Accordingly, the students were asked to submit relevant design documents in UML, code and executables for their project.

Iteration 2.6

The final lecture revisited the issues covered and discussed issues that the students thought were still unclear to them. The students also delivered the final version of their project with the accompanying requirement documents, design documents and API documentation.

CONCLUSIONS

Students generally showed a better appreciation of software engineering methodologies than they did with a traditional course. Most groups carried on improving their designs even after the end of the first half and they were intrigued to find more on software engineering practices on their own. Some of them even redesigned previous projects.

Students also got more chances to develop their problem-solving skills, since they had to experiment with different solutions and find the underlying reasons behind each decision for themselves. They also had to learn to work in a team, to listen to others and sometimes accept that they were not always right.

Compared with previous classes, questions in the second half of the course were more mature and better expressed. Pass rates amongst students that followed the course were also higher. In previous semesters the pass rates of students that were attending the course was around 45% whereas with the new course structure, the pass rate rose to 65%.

The results from a sample of opinions of 25 students that attended the course are presented in Table 2. One student remarked 'The teaching methodology was exponentially better [than the ones in other courses]'. Most students remarked that they found discouraging the small amount of information they had before each class, although they enjoyed the subsequent researching and discussion.

However there are other notes that should also be made.

The presented course structure requires a certain maturity from the students that is not always present in an undergraduate course. This was discovered when attempts were made to follow the same methodology in previous years. Students generally expected the lecturer to first present the 'correct' approach and then ask them to apply this approach to a similar problem. The idea that they had to investigate multiple solutions until they arrived at the 'better' one was some times difficult for them to comprehend.

Students also seemed confused when there was no definite solution to a problem. They would frequently ask at the end of each lecture: 'What shall we write down as the correct answer?' Thus the previous attempts to centre the whole of the semester on this approach were not as successful as students were accustomed to a formally structured course. A similar result regarding students' expec-

Table 2. Students' opinions of the course

Question	Strongly agree	Agree	Neutral	Disagree	Strongly disagree
The adopted course structure helped you in understanding the basic concepts of software engineering? (Requirements capture, design, coupling, cohesion)	80%	20%	0%	0%	0%
The adopted course structure helped you in developing team, communication and problem-solving skills	80%	20%	0%	0%	0%
The adopted course structure should be extended to other modules	60%	4%	36%	0%	0%
The same results with respect to understanding could have been achieved with the traditional course structure	4%	8%	20%	20%	48%
The same results with respect to team, communication and problem-solving skills could have been achieved with the traditional course structure	0%	0%	4%	20%	76%

Table 3. Relative strengths and weaknesses of the proposed approach

Strengths	Weaknesses
Better understanding of crucial concepts in software engineering	Not enough time to cover software management and quality. There should be a follow-up module covering these.
Better understanding of the rationale behind design decisions	Little emphasis on tools
Students actually enjoyed practising the theory	Students must be willing to be active learners. Not always possible in a classroom
Development of problem-solving skills	Difficult to choose an appropriate project
Development of team and communication skills	

tations is also reported in [28], where the authors note: 'The attempt to move towards self-guided learning has been markedly less successful, and students are still relying almost entirely on the lecture to learn the concepts'. The division of the course into two major iterations proved to be a solution to this problem as initially students were let to discover the problems themselves and later to relate the theory being taught to the problems they encountered earlier.

The choice of the initial problem discussed is also important. If the subject matter is too familiar to the students the requirements capture process may be underestimated. If it is, on the other hand, too difficult then too much time may be dedicated to requirements analysis. The design decisions required by the students need to be few in order to narrow down the possible problems that might be encountered but not overly simplistic. In general projects that require the development of a library in the first part and some application of the library in the second part can be useful.

Finally, the adopted course structure did not leave enough time to cover topics such as project management and quality assurance procedures. However these topics are covered in subsequent more advanced modules. The suggested course structure also had little time to spend on the use of modern development tools. Even though this can be considered a drawback, it was done on purpose as it was deemed more important to develop the problem-solving skills of the students rather than spend considerable time on a tool that in the near future may become obsolete.

A summary of the strengths and weaknesses of the proposed approach as the teacher and the students perceived them can be found in Table 3.

Regardless though of the presented weaknesses we believe that the adoption of 'trial-and-error' sessions proved beneficial to the students as the process of software development was de-mystified and better appreciated by them.

REFERENCES

1. Imre Lakatos, *Proofs and Refutations: The Logic of Mathematical Discovery*, (1976).
2. Gerhard Fischer and Matthias Schneider, Knowledge-based communication processes in software engineering, *Proceedings of the 7th International Conference on Software Engineering*, (1984).
3. Mark Priestley, The logic of correctness in software engineering, *1st International Workshop on Philosophical Foundations of Information Systems Engineering*, (2005).

4. Conny Johansson and Lennart Ohlsson, A practice driven approach to software engineering education, *IEEE Transactions on Education*, **38**(5), (1995).
5. J. Tomayko, Carnegie Mellon's software development studio: a five year retrospective, *Ninth Conference on Software Engineering Education*, (1996).
6. Sarah Kuhn, The software design studio: An exploration, *IEEE Software*, **15**(2), (1998).
7. O. Hazzan and Y. Dubinsky, Teaching a software development methodology: the case of extreme programming, *Software Engineering Education and Training*, (2003).
8. Jean Piaget, The psychogenesis of knowledge and its epistemological significance, *Language and learning*, (1980).
9. Lonnie R. Welch, Sherrie Gradin and Karin Sandell, Enhancing engineering education with writing-to-learn and cooperative learning: experiences from a software engineering course, *2002 ASEE Annual Conference & Exposition*, (2002).
10. H. J. C. Ellis, An experience in collaborative learning: Observations of a software engineering course, *Frontiers in Education Conference*, (2000).
11. S. Hadjerrouit, Learner-centered web-based instruction in software engineering, *IEEE Transactions on Education*, **48**(1), (2005).
12. Alex Baker, Emily Oh Navarro and André van der Hoek, An experimental card game for teaching software engineering processes, *Journal of Systems and Software*, **75**(1–2), (2005).
13. David Evans, Teaching Software Engineering Using Lightweight Analysis, http://www.cs.virginia.edu/~evans/pubs/ccli01.pdf, (2001).
14. Daniel Deveaux, Regis Fleurquin and Patrice Frison, Software engineering teaching: a 'Docware' approach, *ACM SIGCSE Bulletin*, **31**(3), (1999).
15. Mehdi Jazayeri, The education of a software engineer, *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, (2004).
16. D. Callahan and B. Pedigo, Educating experienced IT professionals by addressing industry's needs, *IEEE Software*, **19**(5), (2002).
17. R. Conn, Developing software engineers at the C-130J software factory, *IEEE Software*, **19**(5), (2002).
18. Shari Lawrence Pfleeger and Joanne Atlee, *Software Engineering: Theory and Practice*, (2005).
19. Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, (2004).
20. Ian Sommerville, *Software Engineering*, (2004).
21. M. I. Alfonso and F. Mora, Learning software engineering with group work, *Software Engineering Education and Training*, (2003).
22. Carlo Ghezzi and Dino Mandrioli, The challenges of software engineering education, *Proceedings of the 27th International Conference on Software Engineering*, (2005).
23. Barbara Bracken, Progressing from student to professional: the importance and challenges of teaching software engineering, *Journal of Computing Sciences in Colleges*, **19**(2), (2003).
24. M. Gnatz, L. Kof, F. Prilmeier and T. Seifert, A practical approach of teaching Software Engineering, *Software Engineering Education and Training*, (2003).
25. W. G. Bleek, C. Lilienthal and A. Schmolitzky, Weaving experiences from software engineering training in industry into mass university education, *Information Systems Education Journal*, **3**(1), (2005).
26. Apache Software Foundation, Log4J Project, http://logging.apache.org/log4j/docs/index.html, (2002).
27. Frederick P. Brooks, *The Mythical Man Month and Other Essays on Software Engineering*, Pearson, (1995).
28. Stephanie Ludi, Swaminathan Natarajan and Thomas Reichlmayr, An introductory software engineering course that facilitates active learning, *ACM SIGCSE Bulletin*, **37**(1), (2005).

Nicholaos Petalidis is a consulting software engineer. Since 2004 he has been teaching Software Engineering at the Technological Educational Institute of Serres, in the Department of Informatics and Communications. He received his BSc degree in Computer Science from the University of Crete, Greece in 1994 and his Ph.D. from the University of Brighton, UK in 1999. His research interests include software development methodologies and formal methods, but lately he has begun looking into ways to improve the education of future software engineers.