Available online at www.sciencedirect.com
ScienceDirect
Computers & Education 51 (2008) 1007–1016
COMPUTERS & EDUCATION
www.elsevier.com/locate/compedu

An empirical study on students' ability to comprehend design patterns

Alexander Chatzigeorgiou [a,*], Nikolaos Tsantalis [a], Ignatios Deligiannis [b]

[a] Department of Applied Informatics, University of Macedonia, 156 Egnatia Street, 54006 Thessaloniki, Greece
[b] Department of Informatics, Technological Education Institute of Thessaloniki, 54006 Thessaloniki, Greece

Received 29 June 2007; received in revised form 28 September 2007; accepted 7 October 2007

Abstract

Design patterns have become a widely acknowledged software engineering practice and therefore have been incorporated in the curricula of most computer science departments. This paper presents an observational study on students' ability to understand and apply design patterns. Within the context of a postgraduate software engineering course, students had to deliver two versions of a software system; one without and one with design patterns. The former served as a poorly designed system suffering from architectural problems, while the latter served as an improved system where design problems had been solved by appropriate patterns. The experiment allowed the quantitative evaluation of students' preference to patterns. Moreover, it was possible to assess students' ability in relating design problems with patterns and interpreting the impact of patterns on software metrics. The overall goal was to empirically identify ways in which a course on design patterns could be improved.

Keywords: Design patterns; Software engineering course; Empirical study; Teaching methodology; Software metrics; Student assignments

1. Introduction

Design patterns, generally defined as common solutions to common design problems, have been incorporated in the curricula of most computer science and software engineering departments around the world (IEEE Computer Society – ACM, 2001,2004). In such courses students are expected to learn the positive effect of patterns on design quality and become familiar with a variety of design patterns usually found in classic textbooks such as the widely known catalogue by Gamma, Helm, Johnson, and Vlissides (1995). However, there is no general consensus on how to approach the teaching of design patterns given that it is a relatively new field and limited empirical evidence concerning students' perception of patterns is available.

The related literature can be divided into two main categories. The first category contains works that propose courses, assignments and pedagogical frameworks for supporting the teaching/employment of design

* Corresponding author. Tel.: +30 2310 891886; fax: +30 2310 891791.
E-mail addresses: achat@uom.gr (A. Chatzigeorgiou), nikos@java.uom.gr (N. Tsantalis), ignatios@it.teithe.gr (I. Deligiannis).

0360-1315/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compedu.2007.10.003

patterns throughout a computer science curriculum. The second category contains works that evaluate the effect of design patterns on software maintenance.

Astrachan, Berry, Cox, and Mitchener (1998) argued that design patterns facilitate the transition to an object-oriented way of thinking and are essential in learning to use object-oriented techniques correctly and efficiently. Furthermore, they developed pedagogical frameworks, class libraries and expository material for supporting the incorporation of design patterns in computer science curricula. According to Alphonce, Caspersen, and Decker (2007) the teaching of design patterns offers skills and concepts which will be of long-term value to the students even if the underlying technology changes. Gelfand, Goodrich, and Tamassia (1998) proposed the employment of several design patterns in the teaching of data structure algorithms. Cybulski and Linden (2000) developed a multimedia environment for teaching Systems Analysis and Design by a pattern-based learning process that guides students in the effective use of design patterns. Alphonce and Ventura (2002) argued for a design driven approach to the teaching of object-oriented principles. For this reason, they proposed a course sequence that employs design patterns in order to reinforce analysis and design activities at an abstract level. Stuurman and Florijn (2004) presented an assignment for a course on design patterns at the master's level, where students had to adapt an existing program to meet additional requirements. The goal of the assignment was to train students to work with design patterns, as well as to assess that students have reached the learning goals. Pecinovský, Pavlíčková, and Pavlíček (2006) indicated how the *Objects-First* teaching approach can be extended and changed into a *Design-Patterns-First* approach, by presenting the outline of an Object-Oriented Programming course.

The first controlled experiment on the usefulness of design patterns was performed by Prechelt (1997). The experiment aimed to test the hypotheses that software maintainers of well-structured software containing design patterns can make changes faster and with fewer errors if the use of patterns is explicitly documented in the software. The 74 undergraduate students that participated in the experiment were asked to perform pattern-related maintenance tasks on the same program, but separated in two groups. In the first group the program had no pattern documentation, while in the second group additional documentation concerning the use of design patterns was inserted into the program source code. The experimental results had shown that pattern documentation helps software maintainers to be faster and less error-prone in the changes they perform. A replication of the aforementioned experiment was performed by Prechelt, Unger, and Schmidt (1997) having almost the same experimental results. Some of the most important differences between the two experiments are: (a) the participants of the replicated experiment were familiar with more design patterns compared to the participants of the original experiment, (b) the programs that required maintenance were written in C++ instead of Java, (c) the solutions produced by the participants of the replicated experiment could be compiled, while the original experiment was conducted on paper only, (d) the second experiment had only 22 participants instead of 74. Prechelt, Unger, Tichy, Brössler, and Votta (2001) conducted a controlled experiment to compare design pattern solutions to simpler alternatives in terms of maintenance. The subjects of the experiment were professional software engineers that were asked to perform a variety of maintenance tasks. The independent variables were the programs and change tasks, the program version (there were two different functional equivalent versions of each program, a pattern-based version and an alternative version with simpler solutions) and the amount of pattern knowledge of the participants. The dependent variables were the time taken for each maintenance task and the correctness (i.e. whether the solutions fulfilled the requirements of the task). In most of the cases the experimental results had shown positive effects from the use of design patterns, since maintenance time was reduced compared to the simpler alternative versions.

In our study we have attempted to assess students' ability to comprehend design patterns by setting up an experiment based on an assignment given to postgraduate students. In particular, students have been requested to deliver two projects with the same functionality, one with and one without design patterns. The main contribution lies in the fact that this study investigates the usefulness and applicability of design patterns from the students' point of view and at the same time evaluates how well students can comprehend this subject. In this way, the study is also an indirect assessment of the conventional approach in teaching design patterns.

The rest of the paper is organized as follows: Section 2 provides an overview for the experimental set-up discussing the environment in which the study was conducted, the assignment and the participating subject

projects. The results are presented and discussed in Section 3, while threats to the validity of the analysis are mentioned in Section 4. Ideas for future research and improvements for the corresponding courses are suggested in Section 5 and finally we conclude in Section 6.

2. Experimental set-up

2.1. Environment

The experiment on pattern comprehension has been carried out within the context of a course titled ''Advanced Software Engineering''. The course has been running in its present form during the last four years. The course is offered at the postgraduate program of the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. In total, 31 students attended the course during the academic year 2006–2007, separated in two classes taught by the same instructor. All students had undergraduate degrees on computer science/engineering.

To provide insight on students' familiarity with object-oriented software design issues an overview of the course will be described. Advanced Software Engineering aims to present principles and practices for the qualitative and systematic development of large-scale software projects employing object-oriented analysis and design. The course syllabus includes:

- Introduction to Software Engineering (definitions, software project costs, differences from other technical projects, lifecycle models, etc.).
- Crash course on C++ and Java.
- Overview of Unified Modeling Language within the perspective of the Unified Process.
- Object-Oriented Design Principles (Martin, 2003), such as Open-Closed, Single Responsibility, Dependency Inversion, Interface Segregation and Liskov Substitution principles.
- Design Patterns (Gamma et al., 1995)
- Design Heuristics (Riel, 1996).
- Refactorings (Fowler, Beck, Brant, Opdyke, & Roberts, 1999) such as Move field/method, extract method, replace conditional logic with polymorphism, pull up/push down method, etc.
- Software Metrics (size, coupling, cohesion, complexity).
- CASE tool demonstration (forward and reverse engineering capabilities).

Since the experiment focused mainly on the solution of design problems by means of appropriate design patterns, it is worth mentioning the patterns and the sequence by which they have been taught during the course: (1) Adapter (object version), (2) State/Strategy, (3) Template Method, (4) Singleton, (5) Composite, (6) Observer, (7) Bridge, (8) Visitor, (9) Abstract Factory.

The particular patterns and the sequence have been determined according to:

(a) comprehension difficulty (for this reason State/Strategy which employ only an abstraction have been placed at the beginning, while Visitor which involves dual dispatch has been placed at the end).
(b) likelihood of usage in real applications (for this reason patterns such as the Interpreter or Proxy have been excluded).
(c) coverage of all design pattern categories (the above list includes 2 Creational, 3 Structural and 4 Behavioral patterns).

2.2. Assignment description

The experiment on pattern comprehension has been conducted by requesting an obligatory assignment on the implementation of object-oriented software in two versions (having a free choice of subject and implementation language):

- The first (initial) version should not include any design patterns and for this reason the student should document the architectural problems from which it suffers (i.e. design problems that will pose difficulties to the comprehension, testing, maintenance and reuse of software).
- The second (improved) version should resolve the problems of the first version by employing at least two appropriate design patterns. The patterns should be discrete, meaning that they should not be instances of the same pattern. The improvement should be documented by discussing the problems which are solved by each pattern (*qualitative analysis*) as well as by extracting metrics of size, complexity, coupling and cohesion (*quantitative analysis*).

The deliverables of the project were:

1. Functional software for the initial and the improved versions (including an installation and usage manual).
2. Description of functionality (max. one page).
3. Documentation of the architecture by means of UML class diagrams for both versions.
4. Discussion of problems in the first version (max. one page).
5. Description of the design patterns that have been employed in the improved version and discussion of how patterns are related to the specific problems that they solve.
6. Report on the results of the quantitative analysis (metric values per class/package/system) for the initial and the improved version and brief interpretation of the results.

The time given to submit the deliverables of the assignment was six weeks. According to the schedule of the course, during the period in which the first (non-pattern) version was developed (approximately first three weeks), the students had not been introduced to the concept of design patterns.

For the second (pattern) version of the application, the students were free to choose any patterns from the widely known catalog of Gamma et al. (1995) that they considered appropriate, even patterns that have not been covered throughout the course. However, two patterns, namely the object Adapter and the Singleton have been excluded. The reason is that previous experience has shown that students found it easy to apply these patterns without being able to fully justify their necessity.

2.3. Subjects of the experiments

The assignment allowed the students to form teams of two, in order to encourage collaboration in the design and implementation and to provide motivation for increased functionality. However, a number of students preferred to work individually. Information concerning the subject of each project, the number of team members (one or two), the programming language chosen for implementation and the type of user interface (command line or graphical) is summarized in Table 1.

It is worth mentioning that, although students on average had the same level of familiarity with both C++ and Java (based on the courses that the students had attended in previous semesters), almost all teams preferred Java as implementation language. This is probably due to the large number of available APIs in Java and the ease in developing graphical user interfaces that it offers.

From the 31 postgraduate students attending the course on "Advanced Software Engineering", three students did not deliver the assignment and other 5 submitted assignments have been excluded from the analysis because they did not fulfill the requirements (e.g. implementation of only one pattern, initial and improved version had different functionality).

3. Experimental results

3.1. Theoretical basis

According to Basili (1996) our experiment is an in vitro project-based study, with novice participants and descriptive results. According to the level of control it is an observational study since no controlled variables are defined. Although our study is a quasi-experiment in which student teams (experimental units) have con-

Table 1
Project information

ID	Description	No. of team members	Programming language	User interface type
1	Car spare parts management system	2	Java	Graphical
2	Naval engagement	1	C#	Graphical
3	Chess	2	Java	Graphical
4	Body mass index calculator	2	Java	Graphical
5	Video club – movie rentals	2	Java	Graphical
6	FTP client	1	Java	Graphical
7	Car assembly system	2	Java	Graphical
8	ADSL connection upgrade control panel	2	Java	Graphical
9	Accounts department – Salary calculation	2	Java	Command line
10	Disk quota management system	2	Java	Command line
11	File system utility	1	Java	Graphical
12	Wedding reception management system	1	Java	Command line
13	Valve system simulation	1	Java	Command line
14	Airport flights schedule	1	Java	Command line
15	Accounts department – Salary calculation	1	Java	Command line

ducted two software engineering tasks (development of a software system in its pattern and its non-pattern version), it cannot be considered as a controlled experiment. The reason is that the aim is not to compare different populations or processes (treatments) but rather to assess student perception of design patterns (Sjøberg et al., 2005). According to Sjøberg et al. (2005) our experiment can be also considered as a multiple case study since it employs different projects as treatment groups and data is collected as several levels (before and after the introduction of design patterns).

Regarding the assessment of what students have learned from a course that incorporates a novel subject, teaching method or tool, one can find several approaches in the relevant literature: (a) citing indicative comments from students' feedback (Noonan & Hott, 2007), (b) investigation on whether students actually employed the learned topics in their assignments (Noonan & Hott, 2007), (c) evaluation of students performance in exams (Dewan, 2005), (d) statistics concerning perceived learning (Bierre, Ventura, Phelps, & Egert, 2006) and comprehension difficulty (Dewan, 2005) and (e) identification of faults in student project deliverables (Thomasson, Ratcliffe, & Thomas, 2006).

3.2. Descriptive results – discussion

To facilitate the interpretation of the experimental results that will follow, the patterns employed in each project are listed in Table 2.

Table 2
Patterns employed in each project

Project ID	Patterns
1	Bridge, Observer, Visitor
2	Observer, State
3	Abstract Factory, Memento
4	State, Abstract Factory
5	Bridge, Observer
6	Mediator, Command
7	Bridge, Observer, State, Abstract Factory, Composite
8	Bridge, State
9	Bridge, Composite
10	Bridge, Memento
11	Composite, Builder
12	Bridge, Visitor
13	Observer, State, Abstract Factory
14	Observer, Visitor
15	Bridge, Observer, State

A major and common dilemma in all software engineering courses deals with the size of the projects under study: Obviously, within the time limits of an academic semester software projects of a large scale cannot be developed. On the other hand, software engineering principles and particularly object-oriented techniques such as design patterns exhibit their advantages mainly in large scale systems. Fig. 1 shows the diversity of the size of the projects under investigation.

The average project size in the non-pattern and pattern version is 709 and 805 lines of code, respectively, reflecting the expected increase of size when introducing design patterns. A decrease of code size has been observed in only 4 out of 15 projects. All these projects employed the Bridge pattern, which is known to avoid the problem of explosion in the number of classes when abstractions are coupled to implementations (Shalloway & Trott, 2002).

To provide an overview of the popularity of each design pattern among students, Fig. 2 shows the number of projects employing each of the 10 patterns used by the students. Although it is not possible to determine why each pattern has been frequently employed or not, in general, the trend shown in Fig. 2 implies the following: (a) Patterns that have not been taught during the course (Memento, Builder, Mediator and Command) have not been widely used, (b) Patterns that are easy to comprehend and apply (Observer, State) have been frequently used and (c) Patterns that offer solutions in a larger variety of cases (e.g. Bridge, State, Abstract Factory) have been used more often than patterns which handle more specific problems (Memento, Command).

To investigate the ways in which students identified the problems in the software design of the non-pattern version and consequently realized the need for employing design patterns, we asked them to name the prob-

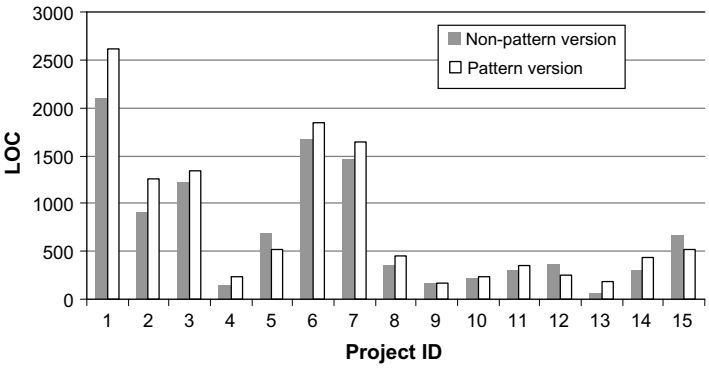

Fig. 1. Size (Lines of Code) for all projects and both versions.

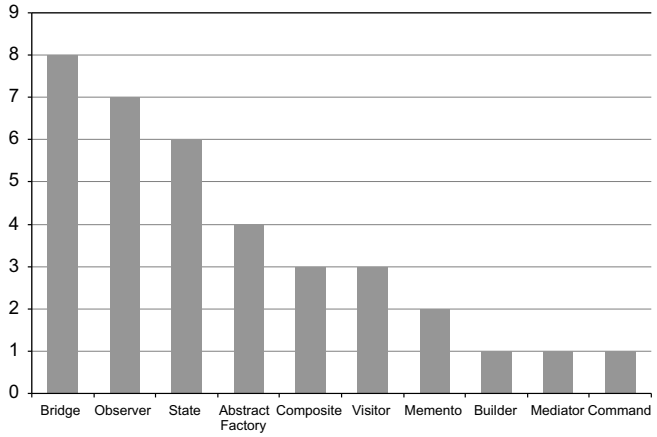

Fig. 2. Number of projects employing each pattern.

lems of the initial software. The questions were open and therefore the categorization shown in Fig. 3 is based on the proximity of each problem to each one of the categories. The results clearly indicate that students perceive as problems in architectural design the maintenance difficulties that a developer will confront if he/she wishes to add new or modify existing functionality. Considering the primary goals of applying design patterns, one of which is to facilitate adaptive maintenance (Ghezzi, Jazayeri, & Mandrioli, 2003), this outcome can be considered encouraging for students learning the value of design patterns in software engineering. By maintainability issues the students primarily identified problems that would arise in the case of specific future requirements. Such problems include the proliferation of classes and the dependency on specific classes, which if changed, would propagate changes to their clients.

The next most common problem reported by the students was code complexity. By code complexity students meant primarily the existence of complex conditional logic, which in general is difficult to comprehend, modify and test. High coupling was identified as the third most common issue in the non-pattern version of the projects. By high coupling students referred to excessive number of associations between system classes, which apart from being difficult to maintain might also lead to ripple effects (Briand, Wuest, & Lounis, 1999).

Finally, in the non-pattern version, the students also identified as problem the existence of God classes and duplicated code. By God class (Riel, 1996) the students named the problem of classes with very large number of responsibilities (i.e. methods). These problems are the least frequently mentioned since design patterns do not target directly at these problems, which can be solved by simpler solutions such as code refactorings (Fowler et al., 1999).

Concerning the problems and the patterns employed to solve them, as reported by the students, the following conclusions can be drawn: Students had difficulties in documenting exactly which problems have been solved by each pattern. This is possibly due to the fact that different patterns may be applicable to the same kind of problems. As a result, the students cannot easily perform a one-to-one mapping between design problems and corresponding patterns, as is the case in other techniques such as bad smells (problems) and refactorings (solutions) (Fowler et al., 1999).

From the well-documented problem-pattern relationships it is worth mentioning the three most frequently reported by the students:

- Maintenance difficulties – Bridge pattern: This is reasonable since students reported the explosion of classes in case of future software enhancements as a typical maintainability problem (Shalloway & Trott, 2002). The Bridge pattern is a typical solution for decoupling implementations from abstractions and thus the extension of functionality is achieved by simply adding new classes.
- Code complexity – State pattern: The State/Strategy design pattern is demonstrated in all classic textbooks on object-oriented design by examples where the use of polymorphism replaces conditional logic in clients. Consequently, this problem–solution relationship appears to be an easy-to-grasp concept for computer science students.

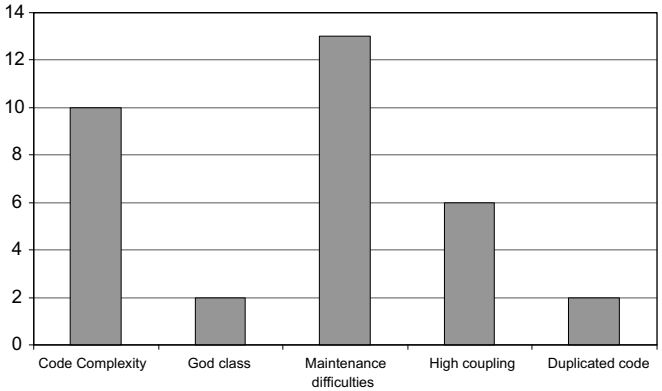

Fig. 3. Number of projects facing each problem.

- High coupling – Observer pattern: In cases where the change in one object (subject) requires the update of a list of dependent objects (observers), the use of the Observer design pattern helps to avoid the tight coupling between subjects and observers. Since students employ similar practices, even unconsciously, as in the case of listeners to GUI events, it was relatively easy for them to apply the Observer pattern and realize its positive effect on coupling.

The assignment required from the students to collect values (for both versions) for the following metrics: Lines of Code (LOC), Number of Classes (NOC), Number of Operations (NOO), Number of Attributes (NOA) (Henderson-Sellers, 1996), Coupling between Objects (CBO), Lack of Cohesion in Methods (LCOM) and Weighted Methods Per Class (WMC) (Chidamber & Kemerer, 1994). Fig. 4 shows the number of student projects exhibiting an increase, decrease or no change for each of the above metrics, comparing the non-pattern and the pattern versions.

As it can be observed from Fig. 4, metrics LOC, NOC, NOO and NOA, which are all size related measures, reflect clearly the increase of size in most cases when design patterns have been introduced. Most students correctly noted in the interpretation of the results for each project that the introduction of patterns was expected to increase the value of size metrics, since design for maintainability requires the addition of new classes (e.g. abstractions and implementing subclasses), which in turn increase code size.

Concerning the CBO metric and particularly the WMC metric, in most projects a positive effect on design quality has been recorded after the introduction of design patterns, i.e. a decrease in the corresponding metric values. In this case most students noted that the reduction of coupling and complexity is a typical result from the application of polymorphism, which is the essence in many design patterns. The improvement in design quality is somehow less intense in the case of cohesion, where cohesion was observed to increase (LCOM metric decrease) in eight out of fifteen projects.

4. Threats to validity

By threats we refer mainly to threats to external validity, encompassing the factors that limit the possibility to generalize our findings beyond the immediate case study to other settings (Yin, 1989). Such factors are:

(a) Instructor's communicability of knowledge and course contents. It is reasonable to expect that students attending the same course in a different environment (i.e. instructor, syllabus) might perceive design patterns more/less effectively and consequently generate different results. This threat is partially confronted by the fact that most courses on design patterns rely on the same classic textbooks and examples.

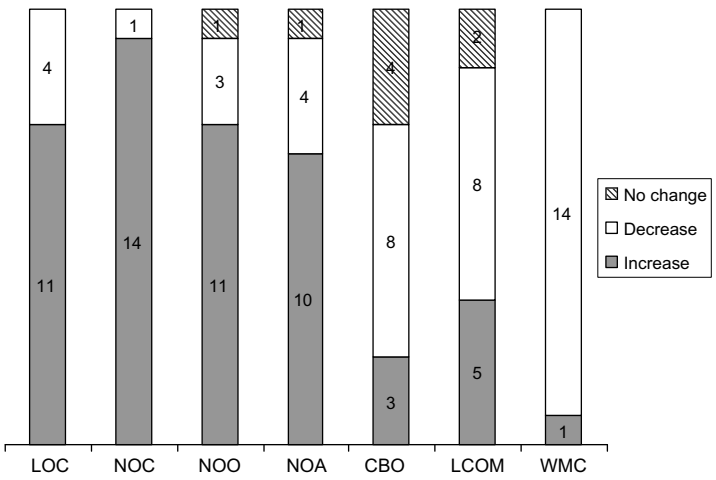

Fig. 4. Number of projects exhibiting increase/decrease/no change for each metric after the introduction of design patterns.

(b) Students' background. Students having a greater programming experience might find it difficult to appreciate the design-before-coding approach promoted by design patterns. This threat is partially alleviated since most students in our study had a relatively medium programming background with a few being exceptionally experienced.
(c) Experimental setting. This threat is valid since students were free to select the subject of their projects. However, some domains are more appropriate for pattern usage than others biasing the results. Moreover, the time given for the completion of the assignment and the number of students per team definitely are factors affecting heavily the results.

5. Future work – Suggestions

This section discusses issues for further research in this field and suggestions for improving a course on design patterns based on the findings of our experiment.

Future research could investigate the evolution of students' projects employing design patterns versus projects without patterns, when adaptive or corrective maintenance is required. In other words the study could focus on the required effort (time, lines of code to be added/modified) and on the modification of design quality when additional requirements are satisfied. Such a study could highlight whether students comprehend the main advantage of employing design patterns which is to facilitate the implementation of future requirements. Moreover, experiments such as the one presented in this work, could be improved by imposing a stricter format in the corresponding assignment. In particular, the assignment could have a fixed report template, project types could be decided by the instructor and functionality could be specified in terms of minimum/maximum function points.

Concerning a software engineering course covering design patterns, the following two student weaknesses that have been observed should be taken into account: (a) students had difficulties in relating design patterns to specific problems that they solve and (b) students found it hard to explain the reason of change in metric values when patterns are introduced. The former problem could be handled by teaching design patterns in analogy to refactorings by presenting them as solutions to certain design problems. This is similar to the approach in (Kerievsky, 2004) where sequences of low-level refactorings are applied to move designs towards pattern implementations in order to handle so-called code smells. The latter problem could be alleviated by discussing the metrics affected by each pattern possibly by illustrating metric value changes in case studies. Currently, in most software engineering courses software metrics are covered separately from design patterns and only general observations are made considering how patterns may affect coupling, cohesion and complexity.

6. Conclusions

The analysis of students' projects involving two versions of a software application has been presented in this paper. One version employed design patterns to solve certain architectural problems while the other did not. The results highlighted several points related to the effective teaching of design patterns in a software engineering course. First of all, it became clear that some patterns are less popular than others indicating that instructors should spend more time on patterns which are intrinsically difficult to understand. Concerning the discussion of design problems in the non-pattern versions, the majority of students correctly identified maintenance problems as the main symptom of a poor architecture. This is in accordance to the general belief that design patterns solve maintenance issues. However, in the discussion of the pattern versions, students found it difficult to relate the applied design patterns to specific design problems and to explain why software metrics changed when patterns have been introduced. This is possibly due to the fact that most patterns are presented in textbooks and courses as solutions to known implementation problems rather than as a treatment for design quality issues. In other words, the design quality improvement, which can be quantified by means of metrics, is presented as a consequence rather than a goal of pattern application. The above observations can be valuable in redesigning the corresponding courses and enhancing design pattern textbooks.

Acknowledgements

This work was funded by the Greek Ministry of Education (25%) and European Union (75%) under the EPEAEK II program ''Archimedes II''.

References

Alphonce, C., Caspersen, M., & Decker, A. (2007). Killer ''killer examples'' for design patterns. In *Proceedings of the 38th SIGCSE technical symposium on computer science education*, pp. 228–232.

Alphonce, C., & Ventura, P. (2002). Object orientation in CS1-CS2 by design. In *Proceedings of the 7th annual conference on innovation and technology in computer science education*, pp. 70–74.

Astrachan, O., Berry, G., Cox, L., & Mitchener, G. (1998). Design patterns: An essential component of CS curricula. In *Proceedings of the 29th SIGCSE technical symposium on computer science education*, pp. 153–160.

Basili, V. R. (1996). The role of experimentation in software engineering: Past, current, and future. In *Proceedings of the 18th international conference on software engineering*, pp. 442–449.

Bierre, K., Ventura, P., Phelps, A., & Egert, C. (2006). Motivating OOP by blowing things up: An exercise in cooperation and competition in an introductory java programming course. In *Proceedings of the 37th SIGCSE technical symposium on computer science education*, pp. 354–358.

Briand, L. C., Wuest, J., & Lounis, H. (1999). Using coupling measurement for impact analysis in object-oriented systems. In: *Proceedings of the 15th international conference on software maintenance*, pp. 475–482.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Cybulski, J. L., & Linden, T. (2000). Learning systems design with UML and patterns. *IEEE Transactions on Education, 43*(4), 372–376.

Dewan, P. (2005). Teaching inter-object design patterns to freshmen. In *Proceedings of the 36th SIGCSE technical symposium on computer science education*, pp. 482–486.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gelfand, N., Goodrich, M. T., & Tamassia, R. (1998). Teaching Data Structure Design Patterns. Proceedings of the 29th SIGCSE technical symposium on computer science education, pp. 331–335.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2003). *Fundamentals of software engineering* (2nd ed.). Prentice Hall.

Henderson-Sellers, B. (1996). *Object-oriented metrics: Measure of complexity*. Prentice Hall.

IEEE Computer Society – ACM (2001). Computing curricula 2001, Computer Science. Available from: http://acm.org/education/curric_vols/cc2001.pdf.

IEEE Computer Society – ACM (2004). Software engineering 2004, Curriculum guidelines for undergraduate degree programs in software engineering. Available from: http://sites.computer.org/ccse/SE2004Volume.pdf.

Kerievsky, J. (2004). *Refactoring to patterns*. Addison-Wesley Professional.

Martin, R. C. (2003). *Agile software development: Principles, patterns and practices*. Prentice Hall.

Noonan, R. E., & Hott, J. R. (2007). A course in software development. In *Proceedings of the 38th SIGCSE technical symposium on computer science education*, pp. 135–139.

Pecinovský, R., Pavlíčková, J., & Pavlíček, L. (2006). Let's modify the objects-first approach into design-patterns-first. In *Proceedings of the 11th annual conference on innovation and technology in computer science education*, pp. 188–192.

Prechelt, L. (1997). An experiment on the usefulness of design patterns: Detailed description and evaluation. University of Karlsruhe, Germany, Technical Report 9/1997.

Prechelt, L., Unger, B., & Schmidt, D. C. (1997). Replication of the first controlled experiment on the usefulness of design patterns: Detailed description and evaluation. Washington University, St. Louis, USA, Technical Report wucs-97-34.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., & Votta, L. G. (2001). A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering, 27*(12), 1134–1144.

Riel, A. J. (1996). *Object-oriented design heuristics*. Addison-Wesley.

Shalloway, A., & Trott, J. R. (2002). *Design patterns explained: A new perspective on object-oriented design*. Addison-Wesley.

Sjøberg, D. I. K., Hannay, J. E., Hansen, O., By Kampenes, V., Karahasanović, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering, 31*(9), 733–753.

Stuurman, S., & Florijn, G. (2004). Experiences with teaching design patterns. In *Proceedings of the 9th annual conference on innovation and technology in computer science education*, pp. 151–155.

Thomasson, B., Ratcliffe, M., & Thomas, L. (2006). Identifying novice difficulties in object-oriented design. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pp. 28–32.

Yin, R. K. (1989). *Case study research: Design and methods*. SAGE.