Exploring the Educational Benefits of Introducing Aspect-Oriented Programming Into a Programming Course

Ivica Boticki, *Member, IEEE*, Marija Katic, *Member, IEEE*, and Sergio Martin, *Member, IEEE*

***Abstract*—This paper explores the educational benefits of introducing the aspect-oriented programming paradigm into a programming course in a study on a sample of 75 undergraduate software engineering students. It discusses how using the aspect-oriented paradigm, in addition to the object-oriented programming paradigm, affects students' programs, their exam results, and their overall perception of the theoretically claimed benefits of aspect-oriented programming. The research methodology, consisting of automating the analysis of student-created computer programs, administering surveys, and collecting exam results, provided an objective measurement of the benefits of the paradigm for novice programmers, as well as evaluating their perception of its usefulness. The results show that the use of aspect-oriented programming as a supplement to object-oriented programming enhances the productivity of novice program code software engineering students and leads to increased understanding of theoretical concepts. Students readily accepted the new paradigm and recognized its benefits.**

***Index Terms*—Aspect-oriented programming (AOP), higher education, programming, programming languages, teaching/learning strategies.**

I. Introduction

PROGRAMMING paradigms date back to the advent of modern computers. Even as early as the 1960s, computer scientists, among them Edsger Dijkstra, gave guidelines on how to do programming [1], [2]. These imperative programming postulates were extended by the popular object-oriented programming (OOP) and influenced the positioning of other programming paradigms, such as declarative programming, in the world of modern software engineering. Software engineering has the dynamism and flexibility to allow for the emergence of new paradigms targeted at filling the gaps in the day-to-day use of imperative, object-oriented, and declarative paradigms. The aspect-oriented programming (AOP) paradigm [3] was introduced as a complement to the object-oriented paradigm in order to improve the overall quality of programs. Logging, exception handling, and authorization checks are examples of tasks that are difficult to solve with OOP, but which can be elegantly solved with the AOP.

As a part of this study, and as part of their course activities, students developed programs using both OOP and AOP. This study examined the impact of using AOP on novice programmers' code, and the impact of acquiring the AOP theoretical concepts and applying these to create programs, on students' learning outcomes and their attitudes toward the paradigm. Apart from analyzing learning outcomes in terms of student-generated artifacts, the study examined whether and how learning outcomes change depending on exposure to AOP, potentially significant data for understanding the effect on learning of solving software engineering problems.

The study was approached by developing a research methodology based on three main pillars: 1) the analysis of student-generated programs; 2) the exploration of classic exam results affected by the use of the aspect-oriented paradigm; and 3) the analysis of survey results indicating students' feelings toward the aspect-oriented paradigm and the educational approach employed. The study participants were 75 undergraduate students enrolled in "Programming Paradigms and Languages," a software engineering course taught as a part of the Computing Study program at the Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia. The students were separated into two groups: 1) an experimental group who used AOP to solve a given task, and 2) a control group who used OOP. Under the three-pillar research methodology, student programs were gathered and analyzed using program code metrics to assess their code quality; this allowed any significant differences in program code between the two groups to be spotted. These findings were then triangulated with standard test results to evaluate the educational effects of using AOP, while the survey results gave student attitudes toward the education approach and the use of AOP.

The remainder of this paper gives an overview of the state of the art in AOP, program code metrics, and OOP education, followed by descriptions of the research questions, research methodology, and data collection. The results are then presented and interpreted, and the findings summarized.

II. Theoretical Background

A. Aspect-Oriented Programming

Aspect-oriented paradigm is based on the concept of *separation of concerns*, first mentioned by Dijkstra [4]. This term refers

Manuscript received February 10, 2012; revised April 28, 2012; accepted June 25, 2012. Date of publication August 03, 2012; date of current version May 01, 2013. This work was supported by the Ministry of Science, Education and Sport, Republic of Croatia, under Projects No. 036-0361983-2019 and 036-0361983-2022.

I. Boticki and M. Katic are with the Department of Applied Computing, University of Zagreb, Faculty of Electrical Engineering and Computing, 10000 Zagreb, Croatia (e-mail: ivica.boticki@fer.hr; marija.katic@fer.hr).

S. Martin is with the Electrical and Computer Engineering Dept., Spanish University for Distance Education (UNED), 28040 Madrid, Spain (e-mail: smartin@ieec.uned.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TE.2012.2209119

TABLE I
RUNNING EXAMPLE OF A ROLE-BASED AUTHORIZATION IMPLEMENTATION WITH OOP

```
public interface ISecurable{
  bool IsUserInRole(User user, string role);
}

public abstract class BusinessObject : ISecurable{
  AccessControlList accessControlList;
  //...
  public bool IsUserInRole(User user, string role){
   if (this.accessControlList.IsUserInRole(user, role))
     return true;
   return false;
  }
}
```

```
public class BusinessDivision: BusinessObject {

 public void EnlistEmployee(Employee em, User user, string role){
  if (this.IsUserInRole(user, role)){
        // ...}
  }

 public Employee GetDirector(User user, string role)
   if (this.IsUserInRole(user, role)){
        // ...}
   //...}
}
```

to the division of a system into modules containing common program features or behaviors, often called concerns. Without this, it might be hard for a developer making a change in program code to identify specific concerns if they are scattered across multiple modules or possibly mixed with code implementing other concerns. The more a programming paradigm supports separation of concerns, the less that concerns are entangled and scattered across programs, which is especially important from the point of view of the maintenance and evolution of software. Separation of concerns is supported by all modern programming paradigms, such as OOP through the decomposition and composition mechanisms. However, these mechanisms support only one dominant dimension of separation at a time. This situation, known as *tyranny of dominant decomposition* [5], always results in some functionalities cross-cutting the decomposition, making it difficult to encapsulate them into a single module. Although the idea of separation of *cross-cutting concerns* appeared much earlier with subject-oriented programming [6] and adaptive programming [7], the aspect-oriented programming paradigm proposed by [3] is much more practical. AOP works because of the mechanisms for orthogonal isolation of so-called cross-cutting concerns into separate modules. As a result, the program code created is more reusable and less tangled. This code consists of two parts: base code that is more purpose-specific and additional code that is represented by *aspects*. An aspect contains advice, additional code executed at a join point, a particular point in the execution of a program, such as method invocation, or field access. An executable program is produced through the process of merging both code parts, often referred to as *weaving aspects into classes*. Weaving rules or *pointcuts* are written in aspects or in the base code, depending on the implementation technology; they specify rules that join points need to satisfy. Overall, this mechanism allows developers to concentrate on business logic and cross-cutting concerns separately.

AOP tools have been developed as extensions to popular OOP languages, such as AspectJ for Java [8], [9] and PostSharp for C# [10]. PostSharp was used for the purpose of the study described here.

1) Need for AOP—An Example: This section presents an example that illustrates the need for AOP when learning object-oriented concepts. The emphasis is on program structures that cannot be easily modeled with OOP, and the possibilities of reworking the code with AOP. Reusing code with AOP differs from reusing code with OOP since AOP allows reuse by simply inserting code into appropriate places at compile or run time. Novice programmers therefore have to pay special attention to features that cannot easily be modeled with OOP, perhaps leading to them having a better understanding of program design concepts.

The example is taken from the PostSharp example suite [10] and adapted to represent two distinct implementations of the same problem. The first version is written in OOP in the C# programming language, as shown in Table I, and the second uses both OOP (C#) and its AOP Postsharp extension, as shown in Table II. Both versions present an implementation of a role-based authorization system allowing users access to the parts of a system that correspond to their associated role. In the first example, a call to the IsUserInRole method must be implemented in each concrete business object, illustrating the repetition of code. In the second version, using AOP, the repeating code is extracted into a separate module (aspect), shown in Table II. Advice is represented with the OnEntry method that gets executed before a specified join point, which is in this case a call to a method marked with the [SecuredOperation(“Manager”)] attribute, representing a pointcut, shown in Table II. Generally, attributes can be applied at the class level, therefore affecting all class methods and avoiding attribute repetition.

B. Program Code Metrics

Independent of specific programming languages, program design principles are well rooted in theory. In his early essays, Dijkstra [1], [2], [4] emphasizes the importance of applying design principles to programming in general; some of these

TABLE II
RUNNING EXAMPLE OF A ROLE-BASED AUTHORIZATION IMPLEMENTATION WITH OOP AND AOP

Base code

```
public interface ISecurable{
  bool IsUserInRole(User user, string role);}

 public abstract class BusinessObject : ISecurable{
  AccessControlList accessControlList;
  //...
  public bool IsUserInRole(User user, string roles){
   if (this.accessControlList.IsUserInRole(user, role))
     return true;
   return false;
  }
}
```

```
public class BusinessDivision : BusinessObject
{
    [SecuredOperation("Manager")]
    public void EnlistEmployee(Employee em){
        //...
    }

    [SecuredOperation("Employee")]
    public Employee GetDirector() {
        // ...
    }
}
```

Additional code represented with aspects

```
public class SecuredOperationAttribute : OnMethodBoundaryAspect
{
    string role;

    public SecuredOperationAttribute(string role) {
        this.role = role;
    }

    public override void OnEntry(MethodExecutionArgs args){
        ISecurable securable = (ISecurable)args.Instance;
        if (!securable.IsUserInRole(Thread.CurrentPrincipal.Identity, this.role)){
            throw new SecurityException("The current user does not have the required permissions.");
        }
    }
}
```

principles were later integrated into the object-oriented programming paradigm. So-called *program code metrics* are used to measure how well these design principles have been applied in building computer programs [11], [12]. These metrics are well-defined tools for measuring program code quality and were an important element of this study's methodology, as shown in Table III.

In addition to the full and short names of program code metrics, Table III gives a short description of the philosophy behind each of them and illustrates how changes in them reflect the overall program code quality.

C. Teaching Object-Oriented Programming

The object-oriented paradigm is considered to be one of today's software engineering standards. Principles such as abstraction, encapsulation, and loose coupling guide software engineers in crafting their programs and systems in an object-oriented way. Instead of first thinking about sequence and flow, software engineers focus on entities and their relationships. Therefore, object-oriented programming is a must in every modern computer science curriculum and is taught using a diverse set of programming languages, such as C++, C#, Java, or Smalltalk. It requires a certain amount of theory to be mastered, which means students are required to learn some general principles prior to engaging with programming [13]. These principles are then used when analyzing domains of problems that need to be solved with the use of OOP.

Although a typical course in object-oriented programming consists of a theoretical part (usually in form of lectures) and laboratory exercises, there are studies on how to help novice programmers learn object-oriented programming. They can be roughly classified into approaches focusing on course design [14]–[16], methodological approaches [17]–[20], tools for scaffolding learning [21]–[24], and studies exploring the suitable background for students taking object-oriented programming courses [13], [25], [26]. Benander in [13] states that learners' backgrounds influence their ability to learn a new OOP language Java and calls for special attention when teaching program design principles to novice programmers. It is then not surprising that the active approach to teaching OOP taken by [17] and [27], consisting of additional teacher guidance in designing software solutions, brings good results. This design-focused and practical approach is also confirmed by a study conducted by [26], which points out that the main pitfalls of learning computer programming boil down to inadequate learning strategies, the lack of practice, and the lack of effort. Therefore, the focus on the design itself cannot replace the iterative process of learning programming since beginners might not be able to fully comprehend complex object-oriented solutions due to their inability to create the appropriate mental representations in time [25], [28]. The gap in achieving close-to-operational knowledge of object-oriented programming can be filled with specific tools with embedded scaffolds [22], but should in general focus on reducing the difficulty and overcoming the challenges of producing quality object-oriented program designs [16]. AOP presents a promising technology in this sense, especially as it is being included in the curriculum of a number of higher education institutions [29].

The evolution of innovation in course design, teaching methods, and teaching tools in software engineering is one of the topics of the SPLASH (previously known as OOPSLA) Educator's Symposium [30]. For almost three decades, academics and practitioners have been exploring and advancing a diverse array of software engineering education topics—ranging from object-oriented programming and test-driven development [31]

to, more recently, parallel programming [32]. The symposium remains highly relevant, summarizing and conveying experience in everyday education and reflecting on key software engineering curriculum topics. Especially important for the study presented here are experiences analyzing novices' programming learning process [33], as well as case studies showing that the general technical complexity of software can be reduced, benefiting students, educators, and clients and leading to simpler, more elegant and efficient solutions. As a part of the SPLASH/OOPSLA educators' initiative, researchers utilize a wide set of methodological approaches, including the analysis (mining) of students' program code, which was also part of this study's methodology [34].

This study builds on the overall consensus that novice programmers should be exposed to a certain amount of practice, as well as theory. The authors also feel that a diverse set of approaches and tools has to be used in order to reduce the difficulty of learning object-oriented programming, and that AOP principles can direct students toward producing better program code. They are also interested in exploring how the practice of using AOP affects students' academic outcomes and how it shapes their perception of the code they produce and the paradigm itself.

III. Courses and Settings

The Department of Applied Computing in the Faculty of Electrical Engineering and Computing, University of Zagreb, delivers pregraduate, graduate, and post-graduate software engineering courses. Course topics range from the introductory (such as the basics of programming in the C programming language for pregraduate students), through the more advanced (such as programming paradigms and languages) to industry-oriented (such as business intelligence and project management at the graduate level).

The focus of this study was a course "Programming Paradigms and Languages," taught to future software engineers in the third year of the pregraduate university level. This course is mandatory for all software engineering students, but is open to other profiles of students. Generally, the course topics can be divided into the three main subgroups: 1) the imperative paradigm; 2) the object-oriented paradigm; and 3) the declarative paradigm. This is the first course in which students learn how to program in the object-oriented paradigm, among others. To complete the course, they are required to master the main theoretical principles, solve programming tasks in the form of laboratory exercises, and pass the exams.

This study only focuses on the middle part of the course, examining the educational benefits of extending the topic of object-oriented programming with AOP. It analyzes the outcomes of laboratory exercises that required students to solve specific engineering problems using AOP. Before the 13-week summer semester of the academic year 2010–2011, minor modifications were made to the course organization, one of the most important being the integration of AOP into laboratory exercises. The 75 enrolled students were separated into two subgroups: 1) the object-oriented, and 2) the aspect-oriented groups. The division into groups was made on the basis of their academic success in the graded activities of the first part of the course (the imperative paradigm) to ensure that each group was of similar academic ability. This approach avoided the *tertium quid* effect of student success potentially biasing the study results.

Both groups had the same laboratory assignment, with the only difference being that the first group had to solve it using OOP only, while the second group had to use OOP, and then also apply the AOP principles to complete the selected subtasks (see the Appendix). Both groups' solutions were expected to have the same functionality; the difference was to be in the internal code structure. Students of the aspect-oriented group were expected to solve the tasks using aspects, advice, and pointcuts, while the object-oriented group had to use classic object-oriented constructs only. Therefore, in this experimental design, the aspect-oriented group can be considered as an experimental group, while the object-oriented group serves as a control group, controlling the effects of the intervention. Throughout the treatment, both groups had the same teacher, which ensured there were no differences in their teaching. The three-pillar research methodology model was devised to gather all the data needed for a thorough analysis of the educational effects of the introduction of AOP, the most important being the changes in how students wrote computer programs, their academic success, and their perception of both the approach employed and AOP.

IV. Research Questions

The purpose of this study is to understand how the introduction of a new programming paradigm in a pregraduate computing curriculum shapes the way students write their programs, how the practice of programming with the aspect-oriented paradigm impacts their academic success, and what is their perception of the educational benefits of the aspect-oriented paradigm. Since AOP was designed to enhance the OOP mechanisms by dealing with the issue of so-called cross-cutting concerns, which should improve program code quality, this study aims to investigate how this approach changed students' programs. Therefore, the following research question was explored.

- How does the quality of code achieved by students using aspect-oriented programming differ from the quality of code achieved by students using only object-oriented programming?

In addition to measuring code quality, also measured is how well this intervention integrates into the course outcomes. The authors were therefore interested to find out whether students who used AOP in their programming tasks achieved better exam results than students who just studied the paradigm in theory. Therefore, the following research question was explored.

- Do students without established stereotypes of object-oriented programming achieve better academic results in their programming tasks when they use aspect-oriented programming?

Finally, the authors were interested in finding out how the two groups of students differ in their opinions of the usefulness of AOP and this intervention. A pre- and a post-survey were conducted to gather students' opinions of program code design, AOP, and alterations in the course contents. Therefore, the following research question was explored.

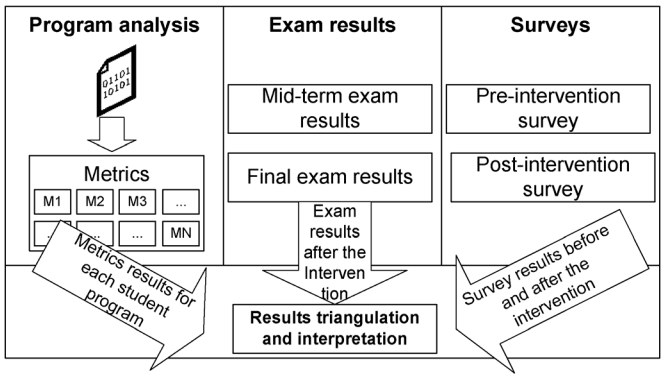

Fig. 1. Three-pillar research methodology model.

- Do students who used aspect-oriented programming in their laboratory exercises have a different opinion as to the usefulness of the aspect-oriented paradigm compared to the students who only used the object-oriented paradigm?

V. Procedure: The Three-Pillar Research Methodology Model

The three-pillar research methodology model, shown in Fig. 1, specifically designed for this study to explore the research questions presented in Section IV, has three main elements, or pillars.

1) In order to understand whether and how the use of AOP affects students' computer programs, these were collected from both the aspect-oriented and the object-oriented groups. These programs were then analyzed using program code metrics, shown in Table III, taking student-generated programs as input and scoring them according to each program code metric as an output. Therefore, each program was given N scores, where N is the total number of metrics used in the research design. In order to figure out the intervention impact, the means of both the experimental and control groups' metrics are compared by an independent sample t-test. This analysis assesses the impact of AOP on the anatomy of student programs.
2) In addition to gaining understanding of how the anatomy of programs changes with the use of AOP, student exam results were collected to determine the effect of AOP on academic success. The AOP is taught in the second part of the course, and there is a midterm exam prior to the third term that consists of written assignments in AOP; these were used to understand how the coding with AOP affects students' understanding of AOP.
3) The third pillar comprises surveys used to get the broader picture of the students' opinions of AOP and the approach employed. Two surveys were administered to both the experimental and the control group—one before, and the other after, the intervention—to collect students' perception of AOP. The authors were interested in finding out how using AOP affects young students' and future software engineers' perception of AOP and program design in general.

Fig. 2. Process of gathering metrics scores for submitted student programs.

VI. Data Collection and Results

A. Student Program Analysis

In order to examine how the use of AOP affects the students' program code, the programs they submitted as their key to the laboratory exercise solutions in the second part of the course were analyzed. This analysis consisted of creating an intermediary program, which takes students' programs as input and examines and evaluates them according to a set of program code metrics, as shown in Fig. 2.

There were 43 aspect-oriented programs submitted $(\mathrm{NA} = 43)$ and 32 object-oriented programs submitted $(\mathrm{NO} = 32)$. These were then fed into a specially designed intermediary program that evaluated them according to a number of metrics. Each row of Table IV presents results of the independent sample t-test, comparing the aspect-oriented and the object-oriented groups according to the metrics results produced by the intermediary program.

The accompanying boxplot diagrams for dependent variables yielding significant between-group differences are shown in Fig. 3. After applying Levene's test for homogeneity of variance and the independent samples t-test, the results show there is a significant difference between groups according to three out of seven program code metrics. For LOC, there is a significant mean difference of $\mathrm{MD} = 160.86$ $(\mathrm{t}(73) = -2.61$, $\mathrm{p} < 0.05$, 2-tailed) with the effect size of $\mathrm{r} = .29$. Both CBM and RFM pass the 2-tailed significance with the mean differences between groups of $\mathrm{MD} = 0.287$ $(\mathrm{t}(73) = -2.15$, $\mathrm{p} < 0.05$, 2-tailed, $\mathrm{r} = .24)$ and $\mathrm{MD} = 1.181$ $((\mathrm{t}(73) = -2.80$, $\mathrm{p} < 0.05$, 2-tailed, $\mathrm{r} = .31)$, respectively. Therefore, there is a medium-sized effect for LOC, CBM, and RFM. An interesting thing to note is that the experimental group achieved better results, ranging from 3% to 22%, when comparing mean values according to all metrics. Although metrics other than CBM, RFM, and LOC do have notable mean between-group differences, they are not significant.

B. Exam Results

An analysis of standard exam results was undertaken to explore whether writing aspect-oriented programs in laboratory exercises translates into academic ability on the standard exam. After submitting their laboratory exercise solutions, students were required to take the second midterm exam, which tested both their practical and theoretical knowledge of OOP and AOP; see Table V.

The table lists the main exams taken by the students during the course. The first midterm exam comparison serves only as a control mechanism in order to confirm there is no significant difference in the academic ability of the groups prior to the intervention $(\mathrm{p} > 0.05)$. Since both the second midterm exam and the final exam contain, in addition to AOP, a diverse set of topics in programming paradigms and languages, a significant

TABLE III
OVERVIEW OF PROGRAM CODE METRICS AND THEIR DESIRABLE DIRECTION OF CHANGE [11]

Metric name	Short name	Metric description[1]	Desirable change
Lines of source code	LOC	Logical number of lines of code containing only lines of executable code and therefore excluding lines such as imports or comments. It is a valuable indicator of the effects of AOP on the overall program code size.	decrease
Weighed operations per module	WOM	Number of implemented methods per module including the information of method complexity. For each method its complexity is calculated according to the number of conditional branches. It affects the size and complexity quality attributes and is important in predicting the time and effort needed for module development.	decrease
Number of children	NOC	Number of derived modules. Since inheritance is a way of reusing code, this metric directly describes reusability meaning that a module with a high number of children has a high degree of reusability. Testability is affected as well because the number of tests increases in case of a high number of children.	Increase
Coupling between modules	CBM	Number of other modules declaring methods or properties used in a measured module. For aspect-oriented programs this measure contains the number of aspects as well. For aspect-oriented programs the metric is calculated with and without aspects being taken into account. If a lot of methods and properties are called from other modules, the measured module is highly coupled.	Decrease
Response for a module	RFM	Number of all module methods and the methods invoked from them. The advice woven into classes is not counted, but the total number of aspects the class uses is. High value means there are many methods in a module, leading to higher module complexity.	Decrease
Depth of inheritance	DIT	The number of base modules in inheritance hierarchy of a module. This metric reveals whether the design is “top heavy” (too many modules near the root) or “bottom heavy” (many modules near the bottom of the hierarchy).	Decrease
Lack of cohesion in operations	LCO	Degree of similarity between methods of the same module based on the usage of instance variables within each method. It describes whether a module lacks cohesion or not. Higher cohesion has a positive impact on encapsulation, while lower cohesion means an increase in code complexity.	increase

[1] Module is a class or an aspect.

difference was not to be expected. However, the fifth assignment in the second midterm exam only tests knowledge of AOP; this shows that the students from the experimental group achieve better academic results than the students from the control group (1-tailed significance with the mean difference between groups of $\mathrm{MD} = 0.171, \mathrm{t}(56) = 1.69, \mathrm{p} < 0.05, \mathrm{r} = 0.22$).

C. Surveys

1) Preintervention Survey: Prior to the intervention, a survey was administered in order to reveal the students’ perception of their own knowledge and their views about certain topics. The survey results clearly show the majority of students think program design is an important topic in software engineering education. It also shows that students know the basics of OOP, but are not familiar with AOP or the related concepts, as shown in Tables VI and VII.

The students’ answers revealed they had been formally introduced to the basics of OOP. Since they were not experienced in software design when they began this course, their believing that it is possible to calculate how well programs are designed is

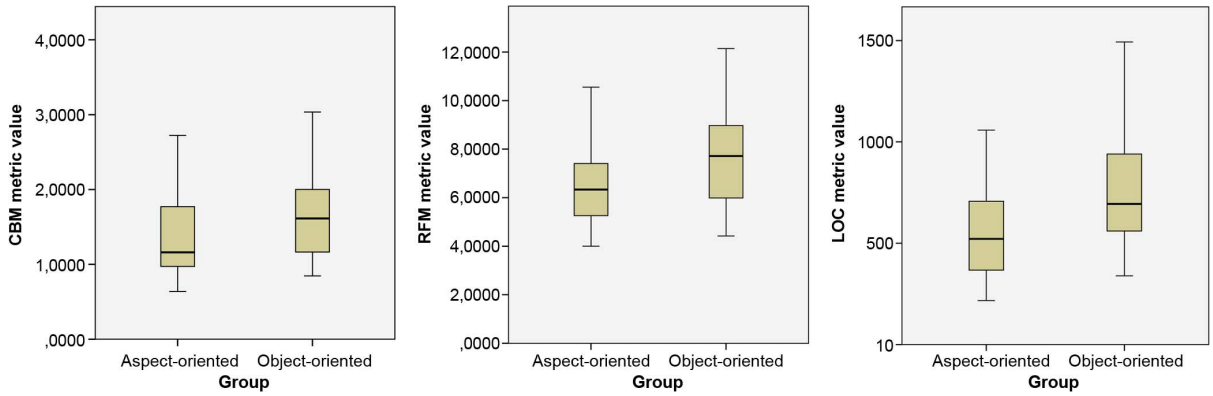

Fig. 3. Boxplots showing dependent variables for CBM, RFM, and LOC metric values across aspect-oriented and object-oriented groups.

TABLE IV
COMPARISON OF THE ASPECT-ORIENTED (EXPERIMENTAL) AND OBJECT-ORIENTED (CONTROL) GROUPS USING THE INDEPENDENT SAMPLE t-TEST (THE CASE OF METRICS)

Metric (desirable change)	Sig. (2-tailed)	Between-group Mean Difference
NOC (-)	0.736	-0.188 (-8%)
CBM (-)	0.035*	-0.287 (-17%)
RFM (-)	0.007*	-1.181 (-15%)
DIT (-)	0.564	-0.336 (-14%)
LOC (-)	0.011*	-160.864 (-22%)
LCO (+)	0.508	33.087 (+19%)
WOM (-)	0.846	-2.124 (-3%)

*$p < 0.05$, 2-tailed

TABLE V
COMPARISON OF THE ASPECT-ORIENTED (EXPERIMENTAL) AND OBJECT-ORIENTED (CONTROL) GROUPS USING THE INDEPENDENT SAMPLE t-TEST (THE CASE OF STANDARD TEST RESULTS)

Metrics	Sig. (2-tailed)	Between-group Mean Difference
1st mid-term exam	0.638	+0.228 (+2%)
2nd mid-term exam	0.426	+0.601 (+6%)
Final exam	0.987	+0.057 (+1%)
5th assignment in the 2 mid-term exam	0.097*†	+0.171 (+28%)

*†$p < 0.05$, 1-tailed

not a surprise. Since they were generally unfamiliar with AOP, the research on this topic was conducted with a high degree of confidence that their prior knowledge would not bias the study results.

2) Post-Intervention Survey: The post-intervention survey was used to gather and compare the experimental and control groups' views of the benefits of AOP. Note that at the time of this survey, the control group had only been introduced to the theory of AOP, but had not done any programming using AOP.

The responses given by the students are predominantly positive toward the use of the AOP. Students think that they are likely to use it in the future, that it helped them in crafting their programs (as part of their laboratory exercises), and that it saved them time while developing programs. What is more, the answers to the questions about program code design had high scores, showing students are convinced that their programs gain in extensibility, reusability, clarity (when reading), and brevity. When the interdependence of components and modularity is concerned, students agree less, but nevertheless are in favor of AOP, as shown in Table VIII.

Interesting results emerge when comparing the experimental and control groups. Students who actually used AOP in programming had a more positive perception of the paradigm. They are generally much more convinced of the benefits of AOP, and the results show the difference is significant ($p < 0.05$). For example, they believe AOP helps them to get work done more quickly and more easily. They also believe the quality of the code they produce is better compared to that of students not using AOP. The AOP group sees the benefits in terms of program extensibility, component interdependence, and modularity. In contrast, there is no between-group difference in the perception of the produced code brevity, reusability, and readability, nor in terms of the perception of object design principle understanding.

VII. CONCLUSION

This study examined the educational benefits of introducing AOP into a programming course. In order to explore how the new paradigm changes the way students create computer programs, and whether and how the practice of using AOP affects students' knowledge, the three-pillar research methodology model was used. The model also provides the means of examining students' perception of the approach employed, as well as their views of the effects of using the new paradigm. Guided by the model, the data from students' programs, exam results, and two surveys were collected.

TABLE VI
INITIAL SURVEY COLLECTING STUDENTS' PERCEPTION OF THEIR KNOWLEDGE AND VIEWS OF PROGRAM DESIGN, OBJECT-ORIENTED AND ASPECT-ORIENTED PROGRAMMING (FIRST PART—ANSWERS ON LIKERT SCALE 1–5)

Question	Mean	Std. Dev.
I think quality program design is an important topic in computing and software engineering	4.59	0.628
I would like to learn more about quality program design in this school	4.20	0.909
I believe it is possible to calculate (automatically or manually) how well a program is designed	3.27	0.930
I know what object-oriented programming is	4.50	0.790
I know what aspect-oriented programming is	1.73	0.943
I know what a join point in the aspect-oriented programming is	1.41	0.860
I know what aspect weaving in aspect-oriented programming is	1.33	0.686
I know what kinds of problems are solved using aspect-oriented programming	1.46	0.849
I apply aspect-oriented programming techniques in programming	1.34	0.741

TABLE VII
INITIAL SURVEY COLLECTING STUDENTS' PERCEPTION OF THEIR KNOWLEDGE AND VIEWS OF PROGRAM DESIGN, OBJECT-ORIENTED AND ASPECT-ORIENTED PROGRAMMING (SECOND PART—YES/NO ANSWERS)

Question	Mean	Std. Dev.
I think quality program design is an important topic in computing and software engineering	4.59	0.628
I would like to learn more about quality program design in this school	4.20	0.909
I believe it is possible to calculate (automatically or manually) how well a program is designed	3.27	0.930
I know what object-oriented programming is	4.50	0.790
I know what aspect-oriented programming is	1.73	0.943
I know what a join point in the aspect-oriented programming is	1.41	0.860
I know what aspect weaving in aspect-oriented programming is	1.33	0.686
I know what kinds of problems are solved using aspect-oriented programming	1.46	0.849
I apply aspect-oriented programming techniques in programming	1.34	0.741

TABLE VIII
FINAL SURVEY RESULTS SHOWING OVERALL STUDENT RESPONSES, SIGNIFICANCE OF THE INDEPENDENT SAMPLE t-TEST AND THE OBSERVED BETWEEN-GROUP MEAN DIFFERENCE (ANSWERS ON A LIKERT SCALE 1–5)

Question	Overall mean	Sig. (2-tailed)	Between-group Mean Difference
It took me a lot of time to master the basics of AOP	2.18	0.969	-0.008
I will use AOP in the future	3.63	0.053*†	0.462
I will use AOP in the third (next) laboratory exercise even though it is not required	2.73	0.015*	0.584
The use of AOP increases the time needed to complete the second laboratory exercise	1.73	0.014*	-0.613
The use of AOP facilitates the completion of the second laboratory exercise	4.29	0.023*	0.552
The use of AOP makes the program from the second laboratory exercise easier to extend	4.19	0.095*†	0.378
The use of AOP makes the program from the second laboratory exercise more reusable	4.23	0.419	0.183
The use of AOP makes the program from the second laboratory exercise easier to read	4.38	0.694	0.080
The use of AOP significantly reduces the number of lines in the program from the second laboratory exercise	4.41	0.867	-0.033
Without AOP it takes more time to complete the second laboratory exercise	4.12	0.056*†	0.514
The use of AOP reduces the interdependence of program components (classes, aspects) and implementation details of other components	3.90	0.039*	0.409
The use of AOP in the second laboratory exercise makes it easier to group relevant functions into separate modules (classes, aspects)	3.87	0.011*	0.524
Without AOP in the second laboratory exercise, it is more difficult to extend the application according to the specification for the third laboratory exercise	3.55	0.416	-0.200
After learning AOP I better understand the principles of object design	2.96	0.741	0.091

*$p < 0.05$, 2-tailed; *†$p < 0.05$, 1-tailed

With the use of AOP, students' program code structure improved, as measured with the program code metrics. The difference between programming with AOP, or with OOP only, ranges from 8% to 22% in favor of AOP and is significant for three metrics: "Lines of code" (LOC), "Coupling between modules" (CBM), and "Response for a module" (RFM). The students who used AOP wrote shorter and more concise code, and their programs were characterized by better modularity. Although the between-group differences for the other four metrics were not significant, a big absolute between-group difference can be noticed, which offers opportunity for future research in the benefits of AOP.

The students' academic results suggest that the practical approach to AOP helped them to achieve better results on the theoretical part of the standard exam, confirming that the practical nature of AOP allows for a better program code structure, and thus to students acquiring a more consistent and thorough understanding of the theory.

The objective program measurement by metrics is confirmed by the post-intervention survey results. The questions querying the interdependence of program components and modularity get only average responses from the control group, but predominantly positive responses from the experimental group. This significant difference shows that students think their designs changed in terms of program code structure, which is in line with the results shown by the metrics, and also confirms that they had become critical enough to realize that the change had actually happened. Although students generally agree the AOP leads to better reusability, readability and brevity, there is no significant difference in perception of these program characteristics between the aspect-oriented and object-oriented groups. At this point, one can only speculate about the reasons, but it might be that complexity and the steep curve of learning programming results in beginners' inability to immediately recognize the benefits and the practical importance of the paradigms and tools used. The observation is in line with the last post-intervention survey item indicating students' perception that not much learning about object design principles happened after learning AOP (both in theory and practice).

There are two possible limitations to the research presented in this paper. The sample size of 75 students means that the results, although significant, come with a medium-effect size. Reproducing the results (with an alpha-error of 0.05 and a power of 80%) might be done with a higher number of participants (e.g., 132). Second, the study covered the short period of one third of a semester, which only gives an overview of the short-term effect of the approach. Clearly, the long-term effects of using AOP still have to be examined and carefully evaluated.

There are plans to continue working on developing the study by designing a Web-based tool integrated into the Moodle online learning environment [35] allowing students to get immediate feedback on the quality of their solutions. This would offer a richer set of experiences in figuring out the relationships and interdependence of program code metric scores and the program receiving the scores. Additionally, the authors would like to strengthen the relationships between the practical and theoretical sides of the student learning experience so they are able to integrate, interpret, and reinterpret theory in the light of their practical experience.

In summary, this study has shown that the use of AOP, as a supplement to OOP, has important educational benefits in terms of improved student program code. It also shows that using AOP in practice leads to increased understanding of the underlying theoretical concepts. Although student perception is in line with these findings, the fact that there is no significant difference in their perceptions of their code (in terms of brevity, reusability, and readability) between those students who did or did not use AOP indicates that the link between the theory and the practice of developing programs requires further strengthening.

Appendix
Description of the Second Laboratory Exercise Assignment

Prior to the laboratory exercises the participating students were given a small Windows application with an already-implemented user interface. After they had implemented the functionalities of viewing of a Google map, centering the map according to a current user position, and drawing the contours of the university buildings, they were required to implement the following functionalities as a part of the second laboratory exercise.

- Extend the current graphic system to support the dynamic selection of different types of geographic maps (i.e., the user selects whether to use Google Maps, Bing Maps, etc.).
- Enable the messaging system so that textual or graphical messages can be left for the chosen location and be visible on the map.
- Undo and redo options when creating or deleting messages must also be enabled.
- Enable logging of all activities, as well as exceptions that occur while the application runs. The log file must contain the names of called methods and exceptions occurred, the location where the class name method or exception is located, and the time it took for the method to complete or when the exception occurred.

Only the students of the experimental group were required to apply AOP concepts whenever possible. They were motivated to do so because they gained more points for functional aspects.

Acknowledgment

The authors would like to thank Shapecrafters, the company producing post-compilation AOP framework PostSharp, for providing a free academic license of the library to be used by the students and authors. They would also like to thank the anonymous IEEE Transactions on Education reviewers for contributing their insightful advice to this paper. Last but not the least, a special acknowledgment goes to M. Kocet, a graduate student with the Faculty of Electrical Engineering and Computing, University of Zagreb, for participating in this study.

References

[1] E. W. Dijkstra, "A case against the GO TO statement," EWD215, Accessed Jun. 7, 2012 [Online]. Available: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html

[2] E. W. Dijkstra, "The humble programmer," EWD340, Accessed Jun. 7, 2012 [Online]. Available: http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. ECOOP*, 1997, LNCS 124, pp. 220–242.

[4] E. W. Dijkstra, *On the Role of Scientific Thought. Selected Writings on Computing: A Personal Perspective*. New York: Springer-Verlag, 1982, pp. 60–66.

[5] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, "N degrees of separation: Multi-dimensional separation of concerns," in *Proc. 21st ICSE*, 1999, pp. 107–119.

[6] W. Harrison and H. Ossher, "Subject-oriented programming (a critique of pure objects)," *Sigplan Notices*, vol. 28, no. 10, pp. 411–428, 1993.

[7] K. J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method With Propagation Patterns*. Boston, MA: PWS, 1996.

[8] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse AspectJ: Aspect-Oriented Programming With AspectJ and the Eclipse AspectJ Development Tools*. Reading, MA: Addison-Wesley, 2004, p. 504.

[9] AspectJ, "Crosscutting objects for better modularity," Eclipse Foundation, Inc., Ottawa, ON, Canada, Accessed Jun. 7, 2012 [Online]. Available: http://www.eclipse.org/aspectj/
[10] PostSharp. ShapeCrafters, Prague, Czech Republic, 2012.
[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[12] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Independence, KY: Course Technology, 1998.
[13] A. Benander, B. Benander, and J. Sang, "Factors related to the difficulty of learning to program in Java—An empirical study of non-novice programmers," *Inf. Softw. Technol.*, vol. 46, no. 2, pp. 99–107, 2004.
[14] D. P. Yeager, "A different kind of programming languages course," in *Proc. 24th ACM SIGPLAN Conf. Companion Object Oriented Program. Syst. Lang. Appl.*, Orlando, FL, 2009, pp. 667–674.
[15] D. West, P. Rostal, and R. P. Gabriel, "Apprenticeship agility in academia," in *Companion 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, San Diego, CA, 2005, pp. 371–374.
[16] J. E. Heliotis, "Easing up on the introductory computer science syllabus: A shift from syntax to concepts," in *Proc. 24th ACM SIGPLAN Conf. Companion Object Oriented Program. Syst. Lang. Appl.*, Orlando, FL, 2009, pp. 683–686.
[17] I. C. Moura and N. v. Hattum-Janssen, "Teaching a CS introductory course: An active approach," *Comput. Educ.*, vol. 56, no. 2, pp. 475–483, 2011.
[18] C. H. Nevison, "From concrete to abstract: The power of generalization," in *Companion 19th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Appl.*, Vancouver, BC, Canada, 2004, pp. 106–108.
[19] J. Borstler, "Improving CRC-card role-play with role-play diagrams," in *Companion 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, San Diego, CA, 2005, pp. 356–364.
[20] C. Gibbs and Y. Coady, "Understanding abstraction: A means of leveling the playing field in CS1?," in *Proc. ACM Int. Conf. Companion Object-Oriented Program. Syst. Lang. Appl. Companion*, Reno/Tahoe, NV, 2010, pp. 169–174.
[21] C. Romero, S. Ventura, and P. de Bra, "Using mobile and web-based computerized tests to evaluate university students," *Comput. Appl. Eng. Educ.*, vol. 17, no. 4, pp. 435–447, 2009.
[22] C. Depradine and G. Gay, "Active participation of integrated development environments in the teaching of object-oriented programming," *Comput. Educ.*, vol. 43, no. 3, pp. 291–298, 2004.
[23] M. Kolling, "The greenfoot programming environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 1–21, 2010.
[24] C. Egert, K. Bierre, A. Phelps, and P. Ventura, "Hello, M.U.P.P.E.T.S.: Using a 3D collaborative virtual environment to motivate fundamental object-oriented learning," in *Companion 21st ACM SIGPLAN Symp. Object-Oriented Program. Syst., Lang., Appl.*, Portland, OR, 2006, pp. 881–886.
[25] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. Corritore, "A comparison of the comprehension of object-oriented and procedural programs by novice programmers," *Interact. Comput.*, vol. 11, no. 3, pp. 255–282, 1999.
[26] N. Hawi, "Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course," *Comput. Educ.*, vol. 54, no. 4, pp. 1127–1136, 2010.
[27] A. Schmolitzky, "A laboratory for teaching object-oriented language and design concepts with teachlets," in *Companion 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, San Diego, CA, 2005, pp. 332–337.
[28] K. Malan and K. Halland, "Examples that can do harm in learning programming," in *Companion 19th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Appl.*, Vancouver, BC, Canada, 2004, pp. 83–87.
[29] Eclipse Foundation, Inc., Ottawa, ON, Canada, "Who's teaching AOSD," Accessed Jun. 7, 2012 [Online]. Available: http://www.eclipse.org/aspectj/teaching.php
[30] Splash, "Splash conference series," Accessed Jun. 7, 2012 [Online]. Available: http://splashcon.org
[31] J. Spacco and W. Pugh, "Helping students appreciate test-driven development (TDD)," in *Companion 21st ACM SIGPLAN Symp. Object-Oriented Program. Syst., Lang., Appl.*, Portland, OR, 2006, pp. 907–913.
[32] N. Anderson, J. Mache, and W. Watson, "Learning CUDA: Lab exercises and experiences," in *Proc. ACM Int. Conf. Companion Object-Oriented Program. Syst. Lang. Appl. Companion*, Reno/Tahoe, NV, 2010, pp. 183–188.
[33] M. E. Caspersen and M. Kolling, "A novice's process of object-oriented programming," in *Companion 21st ACM SIGPLAN Symp. Object-Oriented Program. Syst., Lang., Appl.*, Portland, OR, 2006, pp. 892–900.
[34] W. Poncin, A. Serebrenik, and M. van den Brand, "Mining student capstone projects with FRASR and ProM," in *Proc. ACM Int. Conf. Companion Object-Oriented Program. Syst. Lang. Appl. Companion*, Portland, OR, 2011, pp. 87–96.
[35] Moodle.org, "Moodle learning management system," Accessed Jun. 7, 2012 [Online]. Available: http://moodle.org

Ivica Botički (M'06) was born in Zagreb, Croatia, in 1981. He received the Ph.D. degree in computing from the University of Zagreb, Zagreb, Croatia, in 2009.

He has worked at the Faculty of Electrical Engineering and Computing, University of Zagreb, since 2004 as a Developer, Research Assistant, Research Associate, Lecturer, Researcher, and, since 2012, as an Assistant Professor. He spent the postdoctoral year 2009–2010 with the National Institute of Education, Nanyang Technological University, Singapore. He works as an educational technology consultant. He is an author and coauthor of journal and conference papers on technology-enhanced learning and software engineering.

Marija Katić (M'09) received the M.Sc. degree in mathematics and computer science from the University of Split, Split, Croatia, in 2005, and is currently pursuing the Ph.D. degree in computer science at the University of Zagreb, Zagreb, Croatia.

She has been a Research Assistant with the Department for Applied Computing, University of Zagreb, since 2007. She worked for Siemens, Zagreb, Croatia, as a Software Developer from 2005 to 2007. She is the author and coauthor of several papers published in refereed conference proceedings. Her research interests include software evolution with focus on dynamic evolution, aspect-oriented programming, and software metrics.

Sergio Martin (M'06) was born in Madrid, Spain, in 1980. He received the Ph.D. degree in electrical and computer engineering from the Spanish University for Distance Education (UNED), Madrid, Spain, in 2010.

He has worked as an Assistant Professor with the Electrical and Computer Engineering Department, Industrial School of UNED, since 2007. Since 2002, he has participated in the department's national and international research projects.

Dr. Martin became an Advisory Board member of the Spanish Chapter of the IEEE Education Society in 2009 and of the IEEE Technology Management Council of Spain in 2010. He has received two best thesis awards and four best paper awards.