Informatics in Education, 2007, Vol. 6, No. 1, 53–66

© 2007 *Institute of Mathematics and Informatics, Vilnius*

Exploiting Open-source Projects to Study Software Design

Christopher P. FUHRMAN
Department of Software and IT Engineering
École de technologie, supérieure Université du Québec
1100 Notre Dame Street West, Montreal, Quebec, H3C1K3
e-mail: christopher.fuhrman@etsmtl.ca

Received: December 2006

Abstract. This article presents an approach to using open-source tools and open-source projects to add realistic and practical examples to a course on software design in a professional master's program of software engineering. Students are encouraged to use object-oriented, open-source software projects available on the Internet, and to analyze their design attributes using open-source tools, to hopefully improve their designs using documented design patterns and other design strategies. The proposed approach provides a variety of realistic examples for study, which can vary from semester to semester, without the instructor having to prepare complicated realistic examples or to rely on over-simplified examples in textbooks. Because the course and the approach are relatively new, a quantifiable assessment of the pedagogical approach has not been presented. However, the argument is made that realistic examples provide for better learning, and evidence is provided to show the feasibility of the approach. The instructor's role is more of a mentor than a traditional teacher, as every open-source project is different from a design perspective.

Key words: software design, software design metrics, reverse engineering, open-source software, open-source tools, software design patterns, UML notation, constructivism.

1. Introduction

Software design is said to be a 'wicked' problem (Budgen, 2003), because "it can be characterized as a problem whose form is such that a solution for one of its aspects simply changes the problem". Therefore, it stands to reason that *teaching* software design is also a difficult task. Some of the challenges include teaching design notations, design modeling, design qualities and design methods.

The first-year undergraduate course in software design at the École de technologie supérieure in Montreal is aimed primarily at specific, object-oriented (OO) design problems, for example, those due to high coupling and low cohesion. A common technique to illustrate these problems in a realistic manner is to teach OO design patterns (Gamma, 1995; Larman, 2005). Design patterns are essentially industry-accepted solutions to recurring design problems. Several pedagogical challenges arise when attempting to teach design patterns, particularly when the nature of the problem they are intended to resolve is complex.

First, the instructor has to convince the student that the ‘recurring design problem’ is indeed recurring. Explaining the context of the problem in a single, realistic instance is usually somewhat complex. Explaining (or even finding) another context of the same problem to convince the student that it truly recurs is difficult as well. Often the examples that are given in some of the more recent books on design patterns seem at times contrived and are not always convincing, especially to students with a certain amount of real-world experience who are capable of seeing the nature of the example. It is difficult to present a design problem that is sufficiently complex to be realistic (and convincing) and yet sufficiently simple to explain in a reasonable amount of text.

A second challenge in teaching software design patterns is that the solution always involves abstractions, using software classes that interact in special ways to resolve the problem at hand. Students must be familiar with the UML notation to understand the static and dynamic aspects of a design pattern as it is presented in the literature. Often, software classes found in the design pattern are “pure inventions” (Larman, 2005). That is, they do not represent any concrete real-world concepts, although they sometimes resemble real-world solutions. A pattern considers other classes from the application as abstractions that play a role in the pattern, and which finally must be replaced with the actual classes from the application once the pattern is applied.

Finally, a third challenge is in applying the design pattern to the real-world problem. This involves having the student substitute the software classes from the problem-ridden design into their appropriate abstract roles of the design pattern. Then there is an important step of implementing the changes in the existing application classes, and integrating the “pure invention” classes. Martin Fowler has referred to this aspect of patterns (Fowler, 2003) when he calls them “half baked”.

Perhaps the best way to appreciate a design pattern’s usefulness is to have it be applied to a real-world problem. However, this step entails first diagnosing a problem that the design pattern can potentially correct. In a complex system, it isn’t always obvious where the design problems are. They tend to appear when one tries to modify the system, for example by adding a new functionality. Software maintainability is a function of software design. Extending software is a good exercise to test its design for maintainability, and many design patterns are intended to support easy extension of the software in certain dimensions.

How can such an experience be achieved in a classroom or laboratory situation? It is virtually impossible for individual students to construct a sufficiently complex software system in the span of a 13-week course, to then spot the design problems after trying to modify the software, and finally correct them by applying design patterns. An instructor could prepare and maintain a design for a complex software project, and the results have shown to be interesting from a practical perspective (Blake, 2005) in some courses. However, this requires a significant amount of time and energy on behalf of the instructor to develop the project and to keep its goals realistic. Furthermore, the interests of an instructor, especially one who is a university professor, are not necessarily those of a software developer. The temptation to simplify the project to achieve pedagogical goals is strong, and it will detract from the realism of the project to be studied.

Open-source software is ubiquitous and has established a generally positive reputation in terms of its quality to compete with proprietary products in the areas of software development, software configuration and change management, office automation, databases, Web browsers, etc. However, we consider it in this article as a source of realistic software artifacts, which, because they have been developed by humans, are certain to have design flaws that can be studied, measured, and finally corrected with design patterns.

The goal of the approach presented here is to put students in touch with realistic, large-scale projects, but without the instructor (or the laboratory assistants) having to prepare and maintain these projects. The idea of using sufficiently complex examples to teach OO principles was proposed by Kristen Nygaard in the Comprehensive Object-Oriented Learning (COOL) project (Karahasanovič and Holmboe, 2006).

To accomplish the goal of our approach, we focus on the analysis of open-source projects, using tools that are available through free academic licenses (e.g., Omondo) or open-source licenses (e.g., State of Flow Metrics, Eclipse Metrics, PMD) in an open-source programming environment (e.g., Eclipse). Essentially, these tools are used to help students spot potential design problems, typically through the examination of some design metrics.

Our approach does not seek to validate the usefulness of any particular design metric, as the domain of software metrics is still an evolving area of research. Instead, students use different metrics such as Cyclomatic Complexity (McCabe, 1976), Number of Statements, Instability (Martin, 2003), etc., which are generated by the tools. It is anticipated that these metrics can be indicator of problem areas in the design, although the approach is ad hoc in nature. The inspiration of this comes from the idea of identifying "bad smells" of code (Fowler and Beck, 1999).

Some students in the master's course had their own favorite tools, some of which were academically licensed, some of which were proprietary or have limited trial licenses. As long as they were able to perform the requested tasks, the tools were acceptable.

Pedagogically speaking, the approach presented in this article is constructivist in nature. The instructor fulfills a role as a facilitator rather than a teacher. The learner's uniqueness is reinforced by the freedom to choose a project of interest as well as a set of tools that support the learning tasks.

The results presented in this paper are anecdotal, based on two offerings of a master's level course in software design at the École de technologie supérieure in Montreal, in a professional master's program. They represent the practical (laboratory exercise) aspects of the course only.

2. The Course

The course is part of a master's program that is a "professional" master's program, as opposed to a "research" master's program. It is geared towards allowing students to become better at developing and maintaining software in industry. The course, whose title translates to "Principles and applications of software design," seeks to provide a broader view

of design issues. The topics covered include the role of design in the life cycle of software development, notations and modeling, assessment of design qualities, application frameworks and software architectures. The course had initially been taught by a teaching assistant at our school using Larman (Larman, 2005) as the primary text. However, despite its lack of applied detail, Budgen (Budgen, 2003) has been used most recently, because it provides a broader and more academic perspective, in line with the master's course description. Budgen considers other types of design methods and notations, not just object-orientation, UML, etc.

There are no pre-requisites for the course. However, most of the professional master's students have OO programming (Java, C++, C#) experience, because the professional master's program requires a minimum of a bachelor's degree in computer science followed by two years' professional experience as a software developer prior to admission. The course is 13 weeks of 3-hour lectures, or 39 hours of class, complemented by approximately 96 hours of self-study and final exams, for a total of 135 hours.

The students are allowed to use two double-sided letter-sized pages of hand-written notes for the final exam, whose questions cover the following topics: basic UML modeling and notation, characteristics of an application framework, white-box vs. black-box design abstractions, basic OO design patterns, techniques for formal design, and component-based design.

3. The Practical Exercises

There were three practical exercises, which will be discussed in detail in this paper. As there are no specific hours dedicated to the practical exercises (i.e., no dedicated laboratory sessions), they were presented and discussed during the lecture time, sometimes in a computing laboratory at the university. Practical exercises required individual and group meetings with the instructor, which took place during the time allotted for the course, although not every week. There was an average of 15 students in the course for the two times it was offered, and the meeting times ranged from 5 to 15 minutes, depending on the group and the phase of the exercise. With eight groups, it is possible to hold 10-minute individual meetings within a period of 90 minutes (half a course period). The final deliverable for each exercise was a concise report, with a final 15-minute presentation for the third exercise. Grading of each report was done on several criteria, but most importantly on consistency (e.g., citing of documented research linking an OO measure and a design quality, choice of an improvement motivated by results from metrics, etc.).

3.1. *The Open-Source Tools*

The following open-source tools were proposed and supported for the use in these exercises. The students were permitted to use other similar tools, with the caveat that licensing and technical support were the students' responsibility.

- Eclipse 3 (`www.eclipse.org`) – a very popular Java development environment with plug-in support.

- Omondo (`www.omondo.com`) – an Eclipse plug-in, providing reverse and round-trip engineering for Java and UML diagrams.
- PMD (`pmd.sourceforge.com`) – an Eclipse plug-in, providing static analysis of Java source code.

Similar, open-source tools that were popular among students but will not be shown in detail of this article included the following:

- State of flow Metrics (`http://www.stateofflow.com/projects/16/eclipsemetrics`) – an Eclipse plug-in, providing various OO metrics for Java projects. The metrics are presented in HTML documents, in the form of a set of Web pages that can be browsed for the project in question.
- Metrics (`http://metrics.sourceforge.net/`, Frank Sauer) – an Eclipse plug-in, providing a different set of OO metrics, more oriented towards agile development (Martin, 2003). The metrics can be viewed in near-real time, and are calculated on each compilation, within the Eclipse environment. There is also a dependency viewer.

3.2. *Appreciation of Design Notation through Reverse Engineering*

The first practical exercise was intended to allow students to understand the value of a design notation in a realistic software project. This exercise was done individually, and each student chose an open-source, OO software project. The SourceForge.org Web site was suggested as a resource for finding such projects, but was not mandatory. To assure a certain complexity, the candidate projects were required to have at least twenty software classes. There was no upper-bound on the number of software classes; however, there was a warning that too much complexity could present problems with the tools used for reverse engineering and measurements.

Java projects, as opposed to C++, C#, etc., were encouraged, as our computing labs are set up with Eclipse 3 software and various plug-ins. However, students were permitted to use whatever OO language they wanted, with the caveat that the tool support was their responsibility. Most students chose Java projects, although a small number opted for C++ and C# and managed to find tools to do the reverse engineering.

The students were given one week to identify an open-source project and confirm it with the instructor by email. One hour of the following course period was held in a computing laboratory, configured with Eclipse 3 and the Omondo plug-in, to show how the tools can be used together to reverse engineer Java source code into UML diagrams. This session is primarily a tutorial of the tools, but students were encouraged to do the tutorial with their identified project to save time.

For this exercise, students were asked to generate class diagrams, sequence diagrams and collaboration (communication) diagrams through reverse engineering. Omondo provides its software free to academic institutions via a site-wide academic license. The students were also asked to generate a state diagram in UML using Omondo, but without the help of automated reverse engineering. They were asked to model the state of something in the software, which could be a connection, a transaction, an object, etc.

Fig. 1. Complex class diagram from reverse engineering of Columba project.

When reverse engineering a class diagram with Omondo, the user is first confronted with the complexity of a class diagram, as shown in Fig. 1. In the context of the exercise, students are asked to find a set of "interesting" classes, that is, classes whose relationship has some significance to the design. The students must attempt to understand the nature of the classes and their relationship, in order to come up with a "theme" for their diagram. It is explained that a class diagram with too many classes, attributes and associations can be almost as unreadable as source code. Students spend time trying to find a good set of classes to present, and learn the features of the tools, such as hiding superfluous information from diagrams, as shown in Fig. 2.

Another capability of reverse-engineering tools can be to generate UML diagrams that represent a dynamic aspect of the software, such as sequence diagrams or collaboration (communication) diagrams. An example of a sequence diagram that has been reverse engineered using Omondo in the context of the exercise is shown in Fig. 3. As before, students try to find methods that will show reasonable diagrams. Some complicated Java methods result in very complex, hard-to-understand diagrams.

3.3. *Analysis of Design Attributes*

The second practical exercise involved work in teams of two students. The teams were formed randomly, and each team was asked to choose one of their members' two projects from the first exercise, for which they would assess attributes of the design and try to relate them to qualities of the software. For example, the maintainability (*quality*) of software is (in theory) based on its modularity (*attribute*), its simplicity (*attribute*), etc.

Fig. 2. Meaningful class diagram (XML functionality) of Columba.

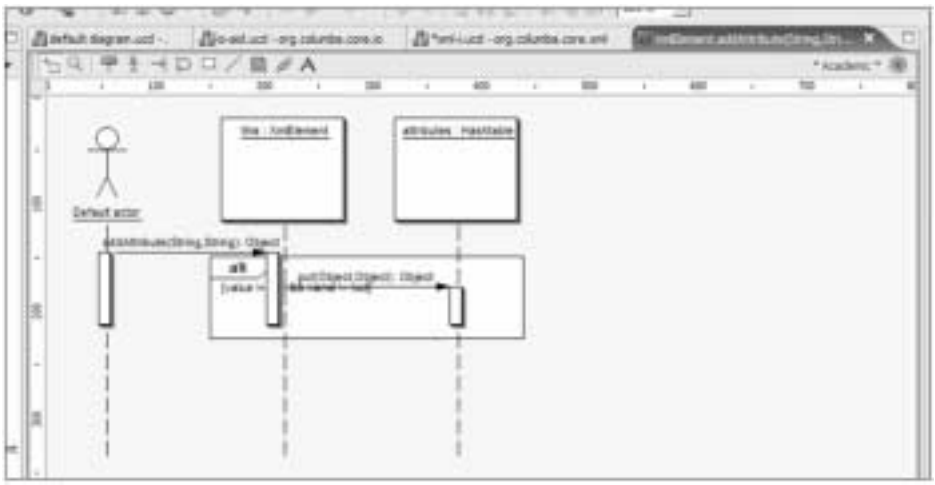

Fig. 3. Example of sequence diagram generated with Omondo.

During the first week of this exercise, to justify the link between the quality and the attribute, they were asked to support the link with evidence published in at least one recent article in a respected scientific journal. This reasoning was discussed in team meetings with the instructor during the following course period. Once some consensus was reached on the qualities and attributes, the team proceeded to measure the attributes using some

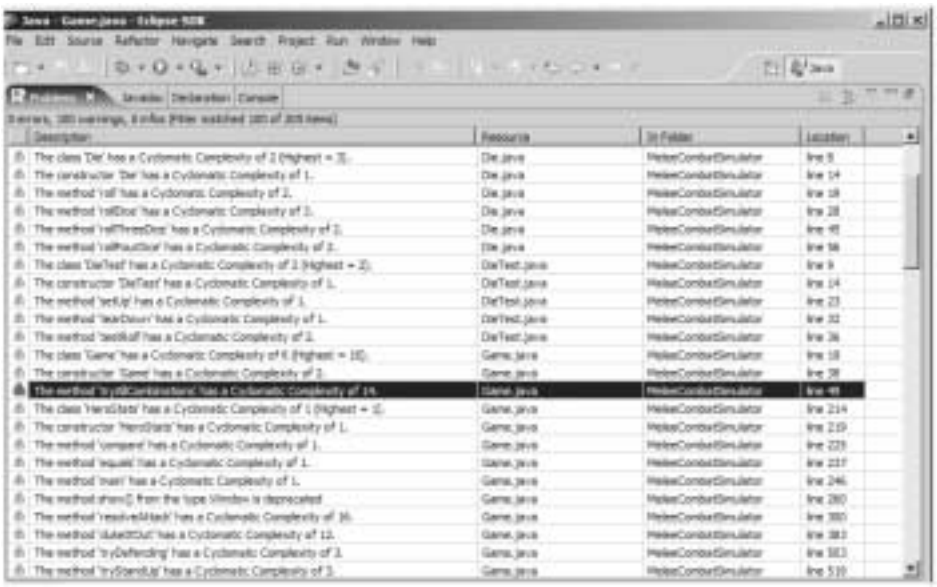

Fig. 4. Sample results from PMD.

method, such as an automated tool, and to reason about the quality in terms of the measured attributes.

For example, if maintainability was linked to modularity, then Coupling Between Objects could be used as a measure. If certain software classes have a high measure of coupling, then it can be reasoned that these classes will in theory be more difficult to maintain. In this exercise, the students are discouraged from discussing in their report any possible strategies to resolve the problems that are identified – this is the goal of the third and final exercise.

As for automated tools to measure design attributes, the students were encouraged to use PMD and Eclipse Metrics, which are flexible tools that scan Java source code and generate messages according to various rules. PMD was already being used in our undergraduate software engineering classes to enforce coding standards, such as proper use of upper- and lower-case in Java class, variable and method names.

Although PMD analyzes primarily the static aspects of Java source code, it can also generate some messages that contain design-oriented measures, such as cyclomatic complexity and coupling between objects. Furthermore, PMD has been adapted to work as a plug-in to Eclipse, so it was deemed a useful and practical tool in this exercise.

Eclipse Metrics has a richer set of metrics, although that does not necessarily mean that all metrics will help in identifying “bad smells” in the design. For example, Lack of Cohesion of Methods (LCOM*) is supported by Eclipse Metrics, but the author of the Eclipse Metrics plug-in himself questions the usefulness of this metric. If a class uses its own getter/setters to access attributes, rather than a direct access to the attributes, it will be penalized with respect to LCOM. As with all metrics, care must be taken before assuming that an anomalous value is an indicator of a design weakness.

3.4. *Design Improvements Proposed by Students*

In this final exercise, the same teams of two students from the previous exercise use the results from that exercise to consider areas to improve the design of their selected project. Because of the varying nature of the projects and the analyses, this exercise has greatest degree of freedom. The only stipulation is that the improvement be motivated by the results of the second exercise (the attribute measures and the related design qualities), and that it be an improvement that is based on strategies documented in the literature.

After receiving their feedback on the second exercise, each team met with the instructor to discuss possible improvements. The instructor's experience in applying design patterns or other strategies to improve a design is essential here, as each project can have very different problems.

Here are examples of improvements that were proposed by students in the course during the autumn semester of 2005:

- improvement of a Model-View-Controller design for better readability;
- use of Polymorphism GRASP pattern (Larman, 2005) to reduce cyclomatic complexity for better maintainability;
- use of State pattern (Gamma, 1995) to reduce cyclomatic complexity for better maintainability;
- use of Façade pattern (Gamma, 1995) to reduce coupling, improve cohesion, for better maintainability;
- use of Chain of Responsibility pattern (Gamma, 1995) to reduce coupling for better maintainability;
- use of Composite pattern (Gamma, 1995) to reduce complexity, class and method length, for better maintainability;
- use of decision tables (Thomas, 2005) to reduce cyclomatic complexity for better maintainability.

In two of the groups, the students followed up with measurements of the same attributes to confirm that there were measurable improvements, although this was not mandatory. In most of the groups, the students implemented their design changes, although this was also not mandatory. Similar results were obtained in the summer semester of 2006.

4. Discussion

4.1. *General*

As this course was taught only one time prior to using this approach, it is difficult to quantify the improvement in learning that took place as a result of the approach involving the evaluation of open-source software. However, we can show that such an approach is certainly feasible from the standpoint of the instructor. Using open-source projects as examples provides thousands of realistic projects, which might potentially pique the

interest of the students more than one or two that come from a textbook. We have also shown that for seven of the projects studied in the autumn, 2005 offering of the course, there were diverse and interesting applications of design patterns to improve the designs based on measurable attributes obtained using open-source tools.

The students generally enjoyed the freedom to choose a project. At the time of writing this article, SourceForge.net had over 140,000 registered projects, of which more than 22,000 were developed in Java, more than 19,000 in C++, and more than 4,900 in C#. However, some students, particularly those less familiar with OO languages, were overwhelmed by this freedom, and needed to be encouraged to choose a project. Many of the students work as software developers in companies in Montreal, and some tried to incorporate projects from their company for the purpose of improvement.

4.2. *First Exercise*

The first exercise on reverse engineering is adequate for exposing the utility of design notations and modeling, and to help the students start to understand the software architecture of the project they have chosen. However, it is not sufficient to teach UML notation, nor is it appropriate for explaining forward engineering. The Larman text (Larman, 2005) is more appropriate for showing how simple UML modeling at a design level can be used to experiment and motivate solutions, before the code is written. This is of equal, if not greater, importance.

Since open-source projects are principally comprised of source-code artifacts, much of the design rationale is impossible to recover with reverse engineering. This limitation was explained explicitly to the students in the handout for exercise #1. However, trying to understand the hidden design rationale using only the source code and reverse-engineered class diagrams of a complex project gives students a better appreciation of why it's important to make clearer the design rationale, either through additional documentation or the use of known design patterns. The initial shock of seeing a complex project from a class-diagram level and trying to make sense of it, can be overwhelming to some students. However, when it is made clear that they are merely to express an important set of classes in their required diagram for the assignment, it is usually reassuring. The task then becomes to try to identify which classes and relationships are important to show in a diagram.

4.3. *Second Exercise*

Regarding the measurements used with design attributes, e.g., complexity, coupling between objects, etc., the PMD plug-in to Eclipse required some adjusting of its threshold values. For example, if all of the classes in a project have measures of complexity below a certain default threshold, no messages will appear in Eclipse from the PMD plug-in. The solution is to set the threshold values for these rules to a number lower than the default (or ultimately 0). Similar arbitrary values exist in the other tools (Eclipse Metrics, State of Flow Metrics).

A limitation with using open-source projects for analysis is that there is generally a lack of explicit specifications of functional or non-functional requirements. The lack of formal requirements can be a problem when considering the evaluation of an open-source design – a design is intended to meet the requirements of the software's specification, either functionally or non-functionally. Without specific requirements, it is difficult to reason about certain qualities of the design. For example, during the second exercise on evaluating the qualities of a design, several groups sought to evaluate the performance of the software and tie it to aspects of the design, which is a very interesting idea. However, a requirement for the second exercise was that the students justify any and all measurements they made with respect to the quality they were evaluating. Since there was no specification of what a "response time" (or other similar performance value) should be for their projects, it could be judged that an evaluation on this basis may be too hypothetical or arbitrary. Similar problems arose when trying to evaluate a design with respect to human-machine interface, reliability, etc.

The design quality evaluation of the second exercise is more likely to be closer to a "white-box" evaluation of the design, since it's more difficult with open-source projects to do a "black-box" evaluation without functional requirements or specification documentation. This limitation brought up a somewhat passionate discussion in the classroom about whether or not open-source projects should include requirements or specification documentation.

According to some of my students who were open-source enthusiasts, providing functional specifications could be considered contrary to the open-source philosophy, as it could be viewed as *imposing* restraints. An open-source project is intended to belong to the community, and people can freely add or change functionalities as they deem appropriate. Any documentation about specifications or requirements that is included in a project might be perceived as too imposing and could possibly drive future contributors away.

4.4. *Third Exercise*

Students generally had little experience with understanding or applying design patterns. In their initial attempts to solve the problems identified in the second exercise, several groups naively applied design patterns to the projects, without really verifying that symptoms of the problem were those of the intended design pattern. Larman refers to this as *pattern-itis* (Larman, 2005). In fact, discussing whether a pattern applies to a set of symptoms is a very good way for the students to learn. However, it can take a lot of time to find a convincing application of a pattern. In one example, a group finally was advised to apply a programming method (decision tables), as opposed to an OO design pattern, because it was difficult to convincingly find a design pattern that resolved the symptoms of a design problem indicated by the measurements.

Hypothetically an extreme scenario is possible, where a software project has no flagrant problems that are indicated by measurable design attributes, or the problems that have been identified cannot be easily resolved using design techniques studied in the

class. If such a case were to occur, the instructor could ask that the student team document a semi-exhaustive analysis of the different design improvements and why they would not apply. Furthermore, in such cases, the students could also be asked to identify the presence of any design patterns that have likely already been applied.

Finally, because they play a role of consultant in this scheme, the instructors need to have a certain practical experience in applying design patterns to different OO projects. The success of the third exercise depends on being able to appropriately apply the correct design patterns, or, conversely, to avoid mis-applying inappropriate design patterns. Much of the pedagogical value of this exercise derives from the discussion about why a pattern fits or doesn't fit. Diligent students usually catch the problems with a mis-applied pattern, but at the very least the instructor has to approve their initial steps on making sure a pattern applies. However, because of the unknown complexity and diversity of a freely chosen open-source project, there is little control over the outcome of this step. But it is exactly this realistic unpredictability that can make the final step in the practical part of this course interesting to the students. The time an instructor may spend developing and maintaining realistic examples can then be spent on mentoring the students on their individual projects.

5. Conclusion

This paper presented some strategies involving the use of open-source tools and open-source projects to effectively teach certain difficult principles of software design in a professional master's program. The strengths and weaknesses have been discussed, from the perspective of the students and the instructor, since a good teaching technique should benefit both parties.

Future alternatives to these methods could focus the choice of open-source projects within a limited design spectrum, for example those developed under certain architectural frameworks (Sun's J2EE, Apache Struts, etc.) to reinforce the architectural and framework aspects of design. Particular design patterns are possibly more likely to be beneficial in these projects.

6. Acknowledgement

Some of the results presented in this article were made possible thanks to the feedback and patience from the students in the MGL802 course in the fall semester of 2005 and the summer semester of 2006 at the ETS. Roger Champagne has also provided useful feedback during the editing of this article.

References

Blake, M.B. (2005). Integrating large-scale group projects and software engineering approaches for early computer science courses. *IEEE Transactions on Education*, **48**, 63–72.

Budgen, D. (2003). *Software Design*. 2nd ed. Harlow, England; Addison-Wesley, New York.
Fowler, M. (2003). Patterns [software patterns]. *IEEE Software*, **20**, 56–57.
Fowler, M., and K. Beck (1999). *Refactoring*: *Improving the Design of Existing Code*. Addison-Wesley, Reading, Mass.
Gamma, E. (1995). *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.
Karahasanovič, A., and C. Holmboe (2006). Challenges of learning object-oriented analysis and design through a modeling-first approach. In A. Fjuk, A. Karahasanovič and J. Kaasbųll (Eds.), *Comprehensive Object-Oriented Learning*: *the Learner's Perspective*. Informing Science Press, Santa Rosa, California, pp. 49–66.
Larman, C. (2005). *Applying UML and Patterns*: *an Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall PTR, Upper Saddle River, N.J.
Martin, R.C. (2003). *Agile Software Development*: *Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, N.J.
McCabe, T.J. (1976). A complexity measure. *Software Engineering, IEEE Transactions*, SE-**2**, 308–320.
Thomas, D. (2005). Agile programming: design to accommodate change. *IEEE Software*, **22**, 14–16.

C.P. Fuhrman is a professor in software engineering at the École de technologie supérieure in Montreal, Quebec, where he teaches undergraduate and graduate courses on OO software design and analysis. In 1988 he obtained a bachelor's degree in computer science at West Virginia University. He later worked for two years as a software engineer at Apple Computer, Inc., on the Apple-DEC Pathworks project. He then spent five years in Lausanne, Switzerland, as a research assistant, earning his PhD at the Swiss Federal Institute of Technology in 1995. He worked for ProLogic, Inc. as the program manager for a research support contract with the NASA Software Independent Verification & Validation Facility in Fairmont, West Virginia. His current research areas are OO software design, agile software processes and lightweight formal methods.

Atvirųjų projektų naudojimas programinės įrangos kūrimui analizuoti

Christopher P. FUHRMAN

Šis straipsnis nagrinėja metodą, kaip naudotis atvirosiomis priemonėmis ir projektais, kad programinės įrangos inžinerijos magistro studijų programos kursą būtų galima papildyti realiais ir praktiškais pavyzdžiais. Besimokantieji skatinami naudotis objektiniais atvirosios programinės įrangos projektais, esančiais internete, analizuoti jų struktūros ypatumus. Siūloma naudotis atvirosiomis priemonėmis, dokumentacija, kitomis strategijomis – šitaip tikimasi pagerinti jomis kuriamas struktūras. Siūlomas metodas teikia daug skirtingų realių pavyzdžių studijoms, jie gali skirtis kiekvienam semestrui. Tad dėstytojas galės be vargo rengti sudėtingus realius pavyzdžius ar pasitikėti supaprastintais pavyzdžiais knygose. Kadangi kursas ir siūlomas metodas gana nauji, tai kiekybinis pedagoginis įvertinimas dar nepristatomas. Vis dėlto manoma, kad realus pavyzdys sąlygoja geresnį mokymąsi – akivaizdu, kad sukonstruotas metodas naudingas ir taikomas. Dėstytojui čia daugiau tenka koordinatoriaus vaidmuo, jis patarinėja, diskutuoja, nes kiekvienas atvirasis projektas paprastai būna vis kitoks.