Learning Software Engineering Principles Using Open Source Software

Jagadeesh Nandigam, Venkat N Gudivada, and Abdelwahab Hamou-Lhadj
nandigaj@gvsu.edu, gudivada@marshall.edu, abdelw@ece.concordia.ca

Abstract - **Traditional lectures espousing software engineering principles hardly engage students' attention due to the fact that students often view software engineering principles as mere academic concepts without a clear understanding of how they can be used in practice. Some of the issues that contribute to this perception include lack of experience in writing and understanding large programs, and lack of opportunities for inspecting and maintaining code written by others. To address these issues, we have worked on a project whose overarching goal is to teach students a subset of basic software engineering principles using source code exploration as the primary mechanism. We attempted to espouse the following software engineering principles and concepts: role of coding conventions and coding style, programming by intention to develop readable and maintainable code, assessing code quality using software metrics, refactoring, and reverse engineering to recover design elements. Student teams have examined the following open source Java code bases: ImageJ, Apache Derby, Apache Lucene, Hibernate, and JUnit. We have used Eclipse IDE and relevant plug-ins in this project.**

Index Terms – Eclipse, Open source software, Source code exploration, Software engineering education.

INTRODUCTION

Teaching software engineering (SE) principles and concepts in a one-semester undergraduate software engineering course is a challenging task. Students often view software engineering principles as mere academic concepts with less practical value. We believe that this perception is a result of the various projects they were involved with in the previous computer science courses such as computer science I, computer science II, data structures and algorithm analysis, where the focus was on mere programming tasks rather than analysis, design, implementation, and maintenance of software systems. As a result, the largest programs that they have written did not exceed few hundred lines of Java code with at most eight to ten classes. Though, they have a conceptual understanding of how to structure a system into subsystems using constructs such as Java packages, they have not applied these concepts due to the trivial nature of the programming problems that they have worked on thus far. In addition, their exposure to Integrated Development Environments (IDE) is limited to the simpler ones such as DrJava and BlueJ, and in some cases it is just a simple editor such as TextPad. They also typically do not get opportunities to examine others' code which prevents them from understanding the challenges of writing code that others can easily comprehend and maintain. Just as reading good writing is used as a basis to teach students to write well, we believe that examining the code written by software professionals entails several benefits for aspiring software engineers.

Therefore, the overarching goal of this project is to teach students a subset of basic software engineering principles by focusing on the practical aspect of software engineering. For this purpose, we selected a subset of activities that we believe can enrich significantly the students experience when dealing with software. These tasks include browsing and exploring an open source code base, assessing the quality of the design based on computed software metrics, applying reverse engineering techniques to synthesize higher abstractions, and performing refactoring and assessing the impact of proposed changes to the code base**.**

The software engineering course where this project is incorporated into the syllabus is offered as a junior/senior undergraduate level course. It should be noted that the intent of our approach is not to provide any quantitative metrics on the tasks that the students were engaged in, but rather to qualitatively assess the effect that the approach had on students' understanding of basic software engineering concepts and principles at both conceptual and pragmatics levels.

The remainder of the paper is organized as follows. In the next section, we describe the SE principles taught to our students. The open source software (OSS) and Eclipse plug-ins that were used in our project are discussed next. Following this, the details of the procedure used in this project are provided. Related work in using open source software in software engineering and computer science education is described next. The last section concludes the paper.

SOFTWARE ENGINEERING PRINCIPLES

We used source code exploration of open source software applications as the primary means to expose students to various software engineering concepts. The SE concepts we focused on in this paper include:

- Familiarizing students with the contents of an open source software applications at various levels of detail using Eclipse IDE perspectives and views.

- Assessing conformance with established coding conventions and style.
- Programming by intention as a way to develop simple, clear, readable, and maintainable code.
- Understanding and appreciating the role of software metrics in assessing code quality.
- Reverse engineering parts of the code to recover various design elements in the form of UML diagrams.
- Understanding what and how of refactoring and applying some basic refactoring techniques to the target source code.

I. Source Code Browsing

Eclipse IDE with JDT (Java Development Tools) [1] and related plug-ins are used for the source code exploration activities of the open source software applications studied. JDT offer four perspectives to understand various parts of a project: Java, Java Browsing, Java Type Hierarchy, and Debug. A perspective in Eclipse workbench is a collection of editors and views. An editor allows you to read and/or write a particular file type. A view is a metadata presentation of information on the workbench. Some commonly used views in JDT include Package Explorer, Outline, and Hierarchy. Using these perspectives and views, one can gain good understanding of the various components that make up the source code of a software package. For most of our students, this is the first time that they have been exposed to software systems that are comprised of various packages and significant number of classes within packages.

II. Coding Conventions and Style

Coding conventions and style contribute to readability and maintainability of an application. In this project, we used coding conventions for the Java language developed by Sun Microsystems. While browsing the source code, students were asked to qualitatively assess the extent to which their code adheres to the Sun coding conventions and consistency in style through manual inspection and/or using a relevant Eclipse plug-in. The goal here is to impress upon the students that coding conventions and style are essential to any real-world software system which is typically developed by scores of software engineers.

III. Programming by Intention

Code readability is enhanced by making your *intention* clear when you write code – *programming by intention* [2]. The main idea of programming by intention is to make code as understandable and intent-revealing as possible by appropriately choosing the names of the identifiers used in a program such as classes, variables, methods, etc. Students recognize that producing quality software is a highly intellectual activity, and never a mechanical task.

IV. Software Metrics

Software metrics measure various aspects of a piece of software [5, 6]. They help characterize features of interest in software quantitatively so to enable classification, comparison, and analysis tasks. In addition, metrics provide support for planning, monitoring, controlling and evaluating the software process and/or product. In this paper, we focus mainly on software product metrics such as size metrics, logic complexity metrics, cohesion, coupling, and certain OO metrics. These metrics can be automatically computed using Eclipse plug-ins. The goal of this activity is to drive home the point that there are quantitative aspects to the software engineering activities.

V. Reverse Engineering

Reverse engineering is the process of analyzing a software system to create representations of the system at higher levels of abstraction [7]. The primary goal of reverse engineering is to help software engineers understand a poorly documented software system when solving maintenance tasks. Reverse engineering has many key objectives, but the ones that are directly relevant to our project consist of generating alternate views and synthesizing higher abstractions from the source code. These views are represented using UML. For this purpose, Eclipse UML plug-ins were used. A secondary goal of this activity is to emphasize the importance of producing and preserving analysis and design artifacts and their role in comprehending large software systems.

VI. Refactoring

The goal of refactoring is to improve the design or internal structure, make it easier to understand and cheaper to modify [2, 3]. Refactoring is an on-going activity during development and should be done when new functionality is added, defects are fixed, code is reviewed, and bad code smells are detected [3]. Major IDEs available in the market today provide some degree of support for performing refactoring activities on source code. Our project uses Eclipse IDE which provides good support for refactoring. Students recognize that just as any good piece of writing requires a few iterations and revisions, producing quality software requires continuous improvements throughout the development life cycle.

OPEN SOURCE SOFTWARE USED

We have used the following open source software packages in the source code exploration project:

- ImageJ – an image processing and analysis system (http://rsb.info.nih.gov/ij/) from the National Institutes of Health (NIH).
- Apache Derby – an open source relational database implemented entirely in Java (http://db.apache.org/derby/). This system has a small footprint that enables it to be embedded into real-time applications.
- Hibernate – an easy to use framework for mapping an object-oriented domain model to a traditional relational database (http://www.hibernate.org/).

- Apache Lucene – is a high-performance, full-featured text search engine library written entirely in Java. It is suitable for nearly any application that requires full-text search (http://lucene.apache.org/java/docs/index.html).
- JUnit – a unit testing framework for the Java programming language (http://www.junit.org/).

The students were also required to download and install the following Eclipse IDE and related plug-ins. Eclipse Plugin Central (http://www.eclipseplugincentral.com) is a central resource that offers the Eclipse community a convenient portal to find useful open source and commercial plug-ins for the entire software development life cycle.

- Eclipse IDE for Java Developers (http://www.eclipse.org/downloads/). This IDE is used for Java development and is crucial for exploring and browsing large source code bases.
- Checkstyle plug-in for Eclipse (http://eclipse-cs.sourceforge.net/). It is a configurable development tool that verifies whether certain Java code adheres to a coding standard. It comes with a default configuration file that supports Sun Java code conventions (http://java.sun.com/docs/codeconv/). Students can define their own coding standards by changing the configuration file.
- Eclipse Metrics 3.3.1 plug-in (http://eclipse-metrics.sourceforge.net/). This tool calculates various metrics such as lines of code, McCabe's cyclomatic complexity, coupling, cohesion, and OO metrics.
- Metrics 1.3.6 plug-in (http://metrics.sourceforge.net/). This tool also calculates various size and logic complexity metrics, cohesion and coupling OO metrics based on the metrics proposed in [8, 9].
- EclipseUML plug-in by Omondo (http://www.eclipseuml.com/), a powerful Java reverse engineering tool that allows visualization of design elements of Java source code. This plug-in is used for reverse engineering activities for the project.
- Refactroing functionality supported within Eclipse IDE.

PROJECT PROCEDURE

The SE project that students worked on used the procedure described in this section. Students worked in teams. Each team is limited to two to three students. Each team is assigned an open source software package from the list – ImageJ, Apache Derby, Hibernate, Apache Lucene, and JUnit. Students were required to complete the following tasks:

- Download and install the latest Java Development Kit (JDK) from Sun.
- Download and install Eclipse IDE for Java Developers.
- Download and install the plug-ins: Checkstyle, Eclipse Metrics 3.3.1 and/or Metrics 1.3.6, and EclipseUML.
- Obtain the source code of the assigned open source software, create a project in Eclipse, and import the source code.
- Each student team uses the IDE and the plug-ins to prepare responses to a generic questionnaire provided to them. Teams submit their project results in the form of a report.
- Each student team is also required to present their project findings at the end of the semester.

Student teams pursue the project with the intent to provide answers to six broad source code exploration activities specified earlier in the Section on Software Engineering Principles. The aim is to provide hands-on learning of basic software engineering concepts. Details of these six activities are briefly discussed in the rest of this section.

I. Source Code Browsing

Teams use Eclipse IDE and its perspectives (Java, Java Browsing, and Java Type Hierarchy) and views (Package Explorer, Outline, and Hierarchy) to obtain answers to the following questions:

- How many packages does your application contain? What are their names?
- Browse the classes within a package using the Package Explorer. Based on the class names and the documentation in the source code, can you tell what the overall purpose of the package is?
- Select any package and one of its classes. Using the Hierarchy View, list all the ancestral classes of the class. Likewise, list all classes that are derived from your chosen class.
- In the Outline View, examine a non-trivial method of a class. Is the code self-describing? Can you explain the purpose of the method without too much difficulty? If not, what factors contributed to un-readability of the code? Are the provided comments useful and seem adequate?

II. Coding Conventions and Style

This activity focuses on coding conventions and coding style. For this purpose, coding conventions for the Java language developed by Sun Microsystems are used. Student teams skim through these coding conventions to answer the following questions:

- Consider the coding conventions for *line length* in the Sun document. Browse your application source code and determine if this coding convention has been consistently followed.
- Consider the coding conventions for *line wrapping* in the Sun document. Browse your application source code and determine if this coding convention has been consistently followed.
- Consider the coding conventions for various *types of comments* – block, single line, trailing, end-of-line, and documentation – in the Sun document. Browse your

application source code and determine if this coding convention has been consistently followed.

- Consider the coding conventions for *declarations* – number per line, placement, and initialization – in the Sun document. Browse your application source code and determine if this coding convention has been consistently followed.
- Consider the coding conventions for *statements* – while, switch, and try-catch – in the Sun document. Browse your application source code and determine if this coding convention has been consistently followed.
- After answering the above questions qualitatively, the project teams ran Checkstyle plug-in on their source code base. Next, they carefully interpreted the results produced by the plug-in with their qualitative assessment of the source code base by visual examination.

III. Programming by Intention

The goal of programming by intention is to develop programs that are clear and readable by choosing names (identifiers) that are semantically transparent. Suggested patterns by Astels [2] for choosing names are - using nouns or noun phrases for class names; using either adjectives or generic nouns and noun phrases for interfaces; using verbs and verb phrases for method names; using accepted conventions for accessors and mutators; and using nouns and noun phrases for variable names. Another suggested pattern by Fowler [3] is not to use comments as "deodorant" and use them for valid and necessary reasons.

To complete this activity, student teams browse at least about 10% of the source code. They browse in a way to cover code across different packages and classes. Teams answer the following questions, and justify their answer by documenting the parts of code visited and the observations made.

- Browse your application source code and determine if class names subscribe to programming by intention principles.
- Browse your application source code and determine if interface names subscribe to programming by intention principles.
- Browse your application source code and determine if method names subscribe to programming by intention principles.
- Browse your application source code and determine if accessor and mutator names subscribe to programming by intention principles.
- Browse your application source code and determine if variable names subscribe to programming by intention principles.

IV. Software Metrics

Using either the Eclipse Metrics 3.3.1 or Metrics 1.3.6 plug-in, student teams compute a set of software metrics for the assigned open source software code base. They comment on the metric values obtained – is metric value within desired range or outside the desired range. If the metric is outside the desired range, suggest what actions (possible refactoring actions) need to be taken. A sample listing of metrics computed using the chosen metrics plug-in is shown below:

- McCabe's Cyclomatic Complexity
- Efferent Coupling
- Afferent Coupling
- Lack of Cohesion in Methods
- Total Lines of Code
- Number of Fields
- Number of Levels
- Number of Parameters
- Number of Statements in Method
- Weighted Methods per Class
- Number of Methods
- Number of Static Methods
- Number of Classes
- Number of Children
- Number of Interfaces
- Depth of Inheritance Tree
- Number of Overridden Methods
- Specialization Index
- Instability
- Abstractness

V. Reverse Engineering

Students conduct reverse engineering activities on the chosen code base using the EclipseUML plug-in. Students are advised to start with a small scope (e.g., a package) to understand how EclipseUML works and then increase the scope to multiple packages or even the entire application. The reverse engineering activities to generate higher abstractions (primarily in graphical representation) of the software system under exploration include:

- Generate a package diagram
- Generate class diagrams
- Generate sequence or collaboration diagrams

VI. Refactoring

Fowler [3] recommends refactoring when *bad smells* are detected in code. Some example bad smells in code include – duplicated code, long method, large class, lazy class, long parameter list, data class, overuse of switch statements, inappropriate intimacy between classes, and unnecessary comments. Effective refactoring requires knowledge of the domain and application code. However, we selected a subset of refactoring techniques that can be performed without much domain knowledge since we do not expect students to be knowledgeable of the domain of the target system. Student teams are encouraged to perform at least the following refactorings without spending too much effort to understand the intimate details of the code base of the open source software package they are exploring.

- Rename Method – If the name of a method does not reveal its purpose, use this refactoring to change the name of the method. Similarly, one can rename classes and fields whose names are not semantically transparent.
- Extract Method – Most of the problems come from methods that are too long. Short, well-named methods are preferred. Fine-grained methods increase likelihood of method reuse. It also allows the higher-level methods (that call the fine-grained methods) to read more like a series of comments. In long methods, there is the problem of semantic distance between the method name and the method body.
- Introduce Explaining Variable – Expressions can become very complex and hard to understand. In such situations, temporary variables can be used to make break an expression into one ore more parts that more readable and manageable.
- Remove Assignments to Parameters – In Java, parameters are *passed by value*. To avoid confusion and increase clarity, it is best not to assign to parameters.
- Replace Magic Number with Symbolic Constant – using magic numbers (literals) are one of the poor practices in computing. Their usage creates code that is less readable and hard to maintain. Replace these magic numbers with symbolic constants.

After these refactorings are performed, students are asked to analyze the impact of each change, and to verify that the change is propagated correctly (i.e., code compiles after each refactoring). Of course, this is only part of the verification process required in any refactoring – proper test cases have to be run before and after each refactoring to verify that observed behavior is still the same.

RELATED WORK

In recent years, there have been several attempts by educators in using open source software for software engineering and computer science education. Carrington and Kim [10] describe how open source software engineering tools are used in a software design course where student teams inspect, report and modify or extend OSS tools to perform reverse engineering, maintenance and refactoring activities. Raj and Kazemian [11] discuss the use of OSS in several advanced courses – database systems, programming language theory, and compilers. Student teams in these courses analyzed source code and made functional enhancements to OSS-based database engines, Scheme tools, and compilers. Pedroni et al. [12] describe how students in an advanced Java course were required to select an open source project, identify parts (bug fixes, extensions, and/or improvements) of the project to contribute code, make changes, and then report their experiences. Nelson and Ng [13] describe a computer networking course that made use of multiple open source packages. O'Hara and Kay [14] describe popular open source licenses and the use of OSS in computer science education and its potential to expose students to larger projects, group work, distributed teams, and peer-review practice. Finally, Jaccheri and Osterlie [15] describe how students can improve their programming and design skills by participating in open source projects.

CONCLUSIONS

Teaching software engineering principles and concepts in a one-semester undergraduate Software Engineering course is a challenging task. Students often view software engineering principles as mere academic concepts with no practical value. To address and overcome these perceptions and related issues, we used a project where we tried to expose students to some of the software engineering concepts in a hands-on manner using open source software in Java, Eclipse IDE and related plug-ins. Each student team is assigned an open source software package to investigate using source code exploration activities.

Student teams pursued the assigned project with the intent to provide answers to six broad source code exploration activities – acquainting with an application, code conventions and style, programming by intention, software metrics, reverse engineering, and refactoring. The teams have documented their findings as responses to a generic questionnaire provided to them by the instructor. The teams have also presented their findings to the class.

The project becomes a compelling exercise if we see its merit qualitatively rather than quantitatively. During the classroom presentations of the project, all student teams unequivocally asserted merit of the project as a means to quickly see the difference between writing a program and developing a software system. We suggest that this project be assigned during the first week of semester and be completed within three weeks.

Overall, the teams have obtained an increased appreciation for software engineering principles and concepts in a very hands-on manner. More specifically, the project activities provided a solid backdrop for the SE course. Subsequent lecture and discussion on design metrics, code maintainability, documentation, keeping the design and code synchronized at all times became more meaningful to the students. For example, for the first time, students could articulate quite eloquently about depth of class hierarchies, and its practical ramifications.

During the course of the project, students have also faced few frustrations that are common and expected when dealing with any real-world open source software. Importing a code base into Eclipse and compiling it was the first hurdle. It required students to discover code dependencies and download related software packages. Instructor's active involvement in the form of scaffolding is quite essential for student success on this project. In summary, this project provided an effective backdrop and platform for students to learn and apply software engineering concepts and principles.

REFERENCES

[1] Valcarcel, C, *Eclipse 3.0 Kick Start*, Sams Publishing, 2005.

[2] Astels, D, *Test-Driven Development – A Practical Guide*, Prentice Hall, 2003.

[3] Fowler, M, *Refactoring – Improving The Design of Existing Code*, Addison-Wesley, 2004.

[4] Booch, G, Rumbaugh, J, and Jacobson, I, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[5] Conte, S, D, Dunsmore, H, E, and Shen, V, Y, *Software Engineering Metrics*, Benjamin Cummings, 1986.

[6] Fenton, N, E, *Software Metrics: A Rigorous Approach*, Chapman and Hall, 1991.

[7] Chikofsky, E, J, and Cross, J, H, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, January 1990, pp. 13-17.

[8] Henderson-Sellers, B. *Object-Oriented Metrics – Measures of Complexity*", Prentice Hall, 1996.

[9] Martin, R, C, *Agile Software Development – Principles, Patterns, and Practices*, Prentice Hall, 2002.

[10] Carrington, D, and Kim, Soon-Kyeong, "Teaching Software Design with Open Source Software," 33rd ASEE/IEEE Frontiers in Education Conference, 2003.

[11] Raj, R, K, and Kazemian, F, "Using Open Source Software in Computer Science Courses," 36th ASEE/IEEE Frontiers in Education Conference, 2006.

[12] Pedroni, M, Bay, T, Oriol, M, and Pedroni, A, "Open Source Projects in Programming Courses," SIGCSE, March 2007.

[13] Nelson, D, and Ng, Y, M, "Teaching Computer Networking using Open Source Software," ITiCSE, 2000, pp. 13-16.

[14] O'Hara, K, J, and Kay, J, S, "Open Source Software and Computer Science Education," The Journal of Computing Sciences in Colleges, Vol. 18, No. 3, February 2003.

[15] Jaccheri, L, and Osterlie, T, "Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering," First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07), 2007.

AUTHOR INFORMATION

Jagadeesh Nandigam, Associate Professor, Computing and Information Systems, Grand Valley State University, Allendale, MI 49401, nandigaj@gvsu.edu

Venkat N Gudivada, Professor, Engineering and Computer Science, Marshall University, Huntington, WV 25755, gudivada@marshall.edu

Abdelwahab Hamou-Lhadj, Assistant Professor, Electrical and Computer Engineering, Concordia University, Montreal, Quebec H3G1M8, Canada, abdelw@ece.concordia.ca