Assignments for an Objects-First Introductory Software Engineering Curriculum

Mathias Ricken
Department of Computer Science, Rice University
mgricken@rice.edu

Abstract

Designing an effective curriculum to teach software engineering to beginning students is challenging. An objects-first course prepares students in an excellent way for the software engineering requirements in industry and academia by focusing on program design, thereby enabling students to write correct, robust, flexible, and extensible software. This paper outlines the effects of an object-oriented approach on software quality and describes three assignments that can be used as teaching tools in an objects-first course to evaluate and reinforce a student's understanding.

1. Introduction

The role of software engineering in computer science education has been a hotly debated topic. Traditionally, programming is taught in an imperative-first style that focuses on programming language syntax and other mechanistic details. Software engineering principles, however, are relegated to upper-level courses.

The ACM/IEEE-CS Report on Computing Curricula 2001 [1] established that introductory courses are often too concerned with language syntax and do not adequately teach design. Because of this focus, imperative-first courses frequently use overly simple assignments that do not motivate students, misrepresent the nature of software engineering, and provide students with too little practice of designing, testing, and documenting their programs. The Software Engineering 2004 Curriculum Guidelines [2] further explain that the programming techniques and processes commonly taught do not scale well past the simple examples used in class.

An alternative objects-first introductory curriculum that combines an early focus on object-oriented (OO) design and programming with unit testing and documentation can alleviate many of the concerns voiced in these reports. Such a curriculum can only be successful, though, if students face interesting and sufficiently complex problems that clearly show the beneficial effects of OO technology on important software engineering quality factors.

2. Software Engineering Quality Factors

Software needs to exhibit certain qualities, which can be external or internal to the program. External factors are visible to a user and include correctness and robustness. Internal quality factors are apparent only to the developer and are important in the implementation and extension of the product. Flexibility, extensibility, testability, and documentation fall into this group. Ultimately, only external quality factors are of immediate interest to a user, but in order to create a program that offers these factors in a timely, cost-effective fashion, developers need to focus on the internal factors as well.

An OO approach to software development can improve the quality factors mentioned above by providing means of managing the program's complexity, thus enhancing the overall

quality of the program. The effects of object orientation on several software engineering quality factors are evaluated below.

2.1. Correctness

Correctness describes the ability of the program to perform its task in accordance with the program specifications and therefore serves as indispensable quality attribute. While it remains difficult to produce large systems without defects, OO design and programming moderate the problem through the use of abstraction and modularity. Both of these principles allow the developer to understand and design parts of the program in isolation.

Abstraction reduces a system's complexity by highlighting commonalities (*invariants*) among similar problems and ignoring differences (*variants*). OO design enables the developer to encode these invariants in concrete portions of a superclass, which deals with the varying portions abstractly and without having to know what they do or how they are implemented. The variants are then placed in subclasses. This separation ensures that invariants cannot be changed, causing the program to be both correct and simpler to understand.

Modularity divides a system into several independent pieces that communicate with each other only via a small, well-defined interface. The use of interfaces as means of communication lets a module deal with other portions of the system at an abstract level, strictly delineating a module's responsibilities. Such loose and abstract coupling between modules limits the knowledge of other objects and thus assists in the maintenance of invariants. A modular design also removes the necessity for developers to be familiar with the entire system by reducing dependencies between portions of the code, thus fostering independent and concurrent development.

2.2. Robustness

Robustness is a system's ability to withstand both accidental and intentional abuse while continuing to function in a well-defined, non-catastrophic manner. Robustness deals with unexpected situations, in contrast to correctness, which derives from system behavior in states covered by the specifications.

Robustness follows from information hiding and a clear delineation of responsibility. The system must be set up to prevent an object from performing actions that are outside its domain, thereby guaranteeing that it cannot adversely affect other components of the system. This guarantee alleviates the need for programmatic value checking in many different places, something easily forgotten when a program grows. Information hiding and delineation of responsibility are design aspects, though; therefore, robustness must be built into the system from the very beginning.

It is hard to design a system that is robust and also flexible and extensible: A design that prevents a system from leaving a well-defined set of states appears resilient to change. If, however, the system makes use of abstraction and modularity, the variants can be changed without undermining the structure enforced by invariant code. Again, the correct decomposition of a problem into variants and invariants leads to the proper abstractions.

2.3. Flexibility and Extensibility

Flexibility and extensibility are important internal quality factors. A flexible system can be changed in one area without breaking other parts or requiring a large number of changes elsewhere. An extensible system can easily be augmented to fulfill new requirements without necessitating changes to the existing portions.

As programs become more complex, the ability to leverage design and code from previous projects and to extend them beyond the original specifications becomes more important. Developers should therefore strive to solve not only a single instance of a problem, but a family of related problems. Again, abstraction and modularity are principles necessary to achieve this. Abstraction allows the developer to generalize a solution for a specific task and apply it to similar ones, and modularity minimizes an object's dependence on other objects with which it interacts, making the object replaceable and reusable.

2.4. Documentation and Testing

Like flexibility and extensibility, documentation and testing are becoming more important as the projects size grows. Unfortunately, internal documentation is often neglected both in software engineering courses and in the industry, even though it is necessary for maintaining a correct and robust solution. The extended life-time of a project gained by flexible and extensible designs further exacerbates this problem, since the behavior of the system must be understood at a later time and often by different developers.

Commenting styles such as Javadoc [3] allow developers to place documentation directly in the code. An external program then automatically processes these comments to create a set of independent and convenient documentation files. This removes the burden of creating the documentation in a separate step and keeps it synchronized with revisions of the code.

Similarly, unit tests can be placed directly in the code to facilitate integration and regression testing. The use of a unit testing framework like JUnit [4] provides an automated way to ensure that a change has no negative effects on the system. Coverage tools like Clover [5] point out program parts that benefit most from additional tests. The two practices of creating unit tests before a program bug is fixed and requiring all unit tests to succeed before code can be committed prevent bugs from reappearing and keep the central repository clean. Unit tests can also be used to provide a more detailed version of the specification and internal documentation, providing both expected results and usage examples.

While documentation and testing are not directly related to an OO approach to software development, their application in the ways described above as part of Extreme Programming [6] is much more prevalent in the OO culture than in the traditional imperative one. It is interesting to note that an imperative-first course, which needs simple examples, to a certain extent *prevents* a focus on flexibility, extensibility, documentation, and testing, while an objects-first course with its demand for sufficiently complex assignments *requires* paying attention to these quality factors.

3. Suitable Assignments

As the previous section has described, OO concepts can improve the quality factors important to software engineering. Unfortunately, many institutions only provide an upper-level course on "OO methodologies". At this time, however, students are unable

to leverage the benefits of object orientation. They possess the intellectual capabilities and the knowledge to use classes, inheritance, and so on, but they fail to see the advantages. This is partially due to the limited time spent on the topic and the small exercises used. To demonstrate the importance of quality factors such as robustness, flexibility, and extensibility, exercises must go beyond small-scale programs.

Finding suitable assignments for an objects-first software engineering curriculum is problematic. The exercises need to demonstrate the positive effect of an OO approach to software quality while remaining within the limited scope of the learning students. Below, we describe three sufficiently complex yet entertaining assignments that can be used in objects-first introductory courses. The Temperature Calculator is intended for an objects-first CS1 course; the other two assignments can be used in CS2 courses.

3.1. Necessary Skills

It is important that the assignments described below are given in the context of a comprehensive instruction in OO design and programming. To successfully complete the assignments, students should possess a solid understanding of inheritance, polymorphism, composition, closures, anonymous inner classes, and delegation model programming. Most of the assignments also require the use of several design patterns, such as composite, visitor, state, singleton, strategy, decorator, template method, abstract factory, and factory method [7]. Since all of the assignments involve GUI programming, the model-view-controller (MVC) pattern is also used extensively.

3.2. Temperature Calculator

The Temperature Calculator assignment [8] consists of a series of applications that convert temperatures from one scale to another. We use this assignment as a 90-minute laboratory exercise that extends into a weeklong homework assignment, which takes students 6 to 8 hours to complete. At the time this assignment is used, students should be acquainted with the factory method, template method, and command patterns.

To illustrate the importance of programming for change—how focusing on internal software quality factors moderates problems with software growth—we guide the students through the development of a program that converts measurements from one unit to another. The programming assignment consists of a series of exercises, each of which imposes a small change in the requirements and forces some appropriate modification on the implementation code. The initial task is the simple conversion of 35° Celsius to Fahrenheit, but the assignment quickly progresses to require dynamically loaded temperature scales that can be compiled later and loaded into the program using reflection. The project culminates in a generalized unit converter that can also process lengths, volumes, and so on. We require the program to be robustly designed and prevent the conversion of, for example, meters to gallons.

To promote code re-use, we apply the Janus Principle [9] and require that the program can support at least two distinct user interfaces: a GUI interface and a command line interface. In many situations, we require the students to modify their code in more than one way and to discuss the advantages and disadvantages.

The incremental nature if the assignment demonstrates how program requirements can change during a project's lifetime, and how flexibility and extensibility can make implementing these changes simpler.

3.3. Games for Two

The Games for Two assignment [10] presents students with a framework for playing two-person turn-based board games on a finite rectangular grid. The project is set up as a GUI application that is flexible and extensible enough to support both human and computer players for arbitrary games that fit the description above. The two examples used in this exercise are Connect–n and Othello.

The project is broken down into two milestones. The first milestone asks students to instantiate the client side of a façade pattern [7] that hides the internals of the game model from players using it. They also need to finish developing the skeleton Connect–n board model by implementing methods to make a move and check its validity. Finally, students should improve the provided random move strategy to only choose from valid moves; the sample solution chooses from all moves, valid or invalid.

For the second milestone, students apply the strategy pattern to write intelligent computer players using the min-max principle, alpha-beta pruning, and depth-limited search. Students are directed towards a high degree of code re-use, which can be achieved by implementing the alpha and beta accumulators as subclasses of the max and min accumulators, respectively. Depth-limited search uses a decorator pattern and can be used in conjunction with any other strategy.

Remarkable about these strategies is that they can be used to play both Connect–n and Othello, and any other game that fits the above description, without modification. The general two-player round-based game model, the board models for the different games, and the movement strategies are loosely coupled and communicate only at an abstract level. In a round-robin fashion, the game model asks the players' strategies for a move and then lets the board model execute it. The search strategies simply look for the best possible move in the current situation, whatever "best" means for a given board model. Due to their generic implementation, the same strategies can be used in all kinds of games. This serves as a convincing example of the powers of OO design.

Since the project is aimed at students at the end of CS2, it requires that students are able to effectively read documentation and analyze an existing framework.

The project also provides students with an opportunity to improve their grades in an entertaining way: At the end of the semester, students can submit specialized Othello strategies and let them compete against each other for extra credit. Every semester, many students participate and voluntarily spend hours fine-tuning their code and thus their programming skills.

3.4. Refactored Marine Biology Simulation

The refactored Marine Biology Simulation [11] was derived from the Java AP Marine Biology Simulation Case Study [12] used in high school AP computer science courses. The original AP case study did not closely follow OO design principles, and it was therefore necessary to refactor it.

Students work on this project over the course of three weeks for a total of about 10 hours. The project places a strong emphasis on creating a robust, flexible, and extensible solution by correctly modeling abstraction and loose coupling. It uses an incremental, test-driven approach that first familiarizes students with the project by extending the framework, which is initially provided as binary code with complementing Javadoc. In the later parts of the assignment, students are asked to re-implement and then improve the framework to achieve additional functionality.

The project provides a framework of 13,000 lines of code and thus possesses the necessary complexity sought for in a final project that is to illuminate the benefits gained from object orientation. Its flexible and robust design was achieved by carefully analyzing the components of the problem and makes use of design patterns such as command, visitor, abstract factory, decorator, and observer-observable [7] to maintain loose and abstract coupling. The assignment therefore places much emphasis on the design aspect of such a system and addresses issues like modeling, components and frameworks, documentation, and testing.

To make sure the considerable size of the project does not overwhelm students, we have split the assignment into two milestones. Milestone 1 requires students to extend the simulation by subclassing while treating most of the framework as a "black box", which serves as an example of how a modular system reduces complexity. In the process of finishing milestone 1, students add a new species of fish and a new type of environment. Both the fish and the environment behave radically different from previous classes, yet the changes require less than 200 lines of code, most of which is boilerplate code.

For milestone 2, students receive the entire framework as source code, which also includes solutions to the problems from milestone 1. Some portions of the framework, however, have been removed and contain only stub code. In the first part of milestone 2, students need to understand the system internals and how the fish and the environment cooperate while remaining decoupled. By re-implementing the portions of the framework that have been stubbed out, students take a grand tour of the system and see how message passing, abstract coupling through interfaces, and several design patterns fit together. The different parts that have been removed were selected to produce a large range of different program failures and thus expose students to different situations requiring debugging: In some cases, a method is not implemented, causing a rupture in the code path, in others a data structure is used improperly, breaking the data flow. Students can use the unit tests provided by the project to continuously monitor their progress and the correctness of their work, while still having to research and implement the system on their own. The test cases offer error messages and hints in plain English but do not reveal the solution.

In the final part of the assignment, students improve the simulation by adding more functionality. This requires changing the fish hierarchy to implement behavior not by inheritance but by delegation. In the course of this exercise, students again experience how separating variants from invariants makes a system both more flexible and less complex.

As a final project for a CS2 course, the assignment presupposes that students have been taught the basic OO skills mentioned in Section 3.1, as well as additional design patterns, especially command and factory method [7]. Design, documentation and testing tools, such as UML, Javadoc and JUnit, respectively, must also be addressed.

Instructors will need to ensure their students understand the concept of a "local environment" and how it is modeled. Students also inquired about the callback-style communication between a fish and its environment, which provides the opportunity to use UML sequence diagrams.

4. An Evaluation of Objects-First

In an attempt to study the quality of education in our objects-first CS1/CS2 courses, we analyzed student performance in a systems class. Specifically, we compared the grades of students having taken the objects-first courses to those of graduate students.

Since we usually require undergraduate transfer students to take our CS1/CS2 courses, graduate students are the only group of students that is largely unaffected by the objects-first approach. The systems course is typically taken right after CS2, uses C and assembly as programming languages, and employs no OO concepts at all.

Over the course of five years, we evaluated the grades of 180 undergraduate and 32 graduate students. We found that the average course grade for undergraduates was 2.97 ("B"; $\sigma \approx 1.138$), the average for graduates was 3.29 ("B+"; $\sigma \approx 0.824$). This means that graduate students outperformed undergraduates by about 12 percent, or a third of a letter grade. Considering that many of the undergraduates were only in their third semester, though, this difference is small. We therefore believe that our objects-first CS1/CS2 courses equip students well even for assignments in more traditional settings. Unfortunately, due to the small size of our institution, it is difficult to obtain data that significantly supports or rejects this hypothesis. Our inquiry also ignores other factors, such as the background of graduate students and prior programming experience.

Reservations against objects-first courses often stem from the view of object orientation as an "advanced" concept too complicated to teach to beginning students. The experience at our institution over the past years and data collected by Phil Ventura at SUNY Buffalo [13], however, suggest the opposite. The main predictor for success in an objects-first curriculum seems to be the effort students put into the class. In Ventura's study, labs attended and exams taken accounted for 86.3 percent of the variance in course scores. Other factors, such as year in college, declared major, GPA, or mathematics background showed only little predictive power. The study also showed that, in contrast to typical imperative-first courses, the evaluated objects-first course is not gender-biased and does not disadvantage students without prior programming experience. The evidence therefore suggests that objects-first is an introduction to computer science for everyone.

Imperative-first courses largely ignore software design, leaving students unequipped when dealing with the larger projects their careers hold in store. Objects-first courses, on the other hand, spend much more time on the design aspects and introduce a language's syntax only when it is necessary. Concepts like classes, inheritance, composition, UML, and some design patterns are usually introduced in the first half of CS1 already. While procedural elements of programming are still introduced towards the end of the first semester, students find this paradigm shift much easier to master than the one found in imperative-first courses [14].

The size and nature of the assignments used in objects-first introductory courses enable students to effectively participate in upper-level courses as well as in the development of complicated large-scale software. In production programming courses, for instance, students have employed OO design and unit testing to develop, maintain, and enhance an integrated development environment (IDE) of considerable complexity (>250,000 lines of code). This enterprise has reportedly been successful over the course of several years in spite of an ever-changing group of student developers who often join directly after having completed the CS2 course [15].

Suitable examples for an objects-first curriculum also lend themselves well for introducing advanced concepts in computer science, such as systems security, graphics, and distributed computing. To demonstrate the steps necessary to achieve robustness, for example, an operating system analogy can be used: Only the kernel can perform certain tasks, and any communication between user processes has to be done using the kernel. Operating systems also provide interfaces that allow user processes to treat devices abstractly; the concrete implementations are hidden in device drivers.

Delineation of responsibilities and abstract device access make an operating system both robust and extensible. In computer graphics, a raytracer can be developed using OO principles: Different geometric primitives form an inheritance hierarchy. In distributed programming, a fork-join framework [16] makes many examples accessible, particularly in combination with a sorting framework [17] using Merrit's taxonomy of sorting algorithms [18].

While these concepts cannot and should not be the focus of an introductory course, they serve to captivate the students' fascination and motivate them to pursue their computer science careers with even greater interest.

5. Conclusion

Both anecdotal and statistical evidence suggests that an objects-first curriculum provides a superb alternative to the prevalent imperative-first approaches, particularly as an introduction to software engineering. For such a curriculum to be successful, though, compelling examples and assignments of adequate complexity are required. The three assignments presented in this paper stress OO design and the resulting robustness, flexibility, and extensibility that would be hard to achieve without the use of object orientation.

The result of using an objects-first approach is an introductory curriculum that addresses several of the problems mentioned in the ACM/IEEE-CS reports and that provides students with the software engineering skills necessary to write better software.

6. References

[1] ACM/IEEE. *Final Report of the Joint ACM/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science*. ACM/IEEE, 2001.

[2] ACM/IEEE. *Software Engineering 2004: ACM/IEEE-CS Guidelines for Undergraduate Programs in Software Engineering*. ACM ICSE 2005.

[3] Sun Microsystems, Inc. http://java.sun.com/j2se/javadoc/

[4] JUnit Project. http://www.junit.org/

[5] Clover. Atlassian Pty Ltd. http://www.atlassian.com/software/clover/

[6] Jeffries, R. http://www.xprogramming.com/

[7] Gamma, E., et al.. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[8] Nguyen, D., and M. Ricken. *Nifty Assignments: Programming for Change*. ACM OOPSLA Educators' Symposium 2006.

[9] Adams, J. *OOP and the Janus Principle*. ACM SIGCSE 2006.

[10] Nguyen, D., and S. Wong. *Design Patterns for Games*, ACM SIGCSE 2002.

[11] Cheng, E., D. Nguyen, M. Ricken, and S. Wong. *Nifty Assignments: Marine Biology Simulation*. ACM OOPSLA Educators' Symposium 2004.

[12] Brady, A. *AP marine biology simulation case study*. J. Comput. Small Coll. 18, 1 (Oct. 2002), 113-114.

[13] Ventura, P. *On the Origin of Programmers: Identifying Predictors of Success for an Objects-First CS1*, Dissertation, SUNY Buffalo, 2003.

[14] Alphonce, C. and P. Ventura. *Object Orientation in CS1-CS2 by Design*, ACM ITiCSE 2002.

[15] Allen, E., R. Cartwright, and C. Reis. *Production Programming in the Classroom*. ACM SIGCSE 2003

[16] Lea, D. *A Java fork/join framework*. In Proceedings of the ACM 2000 Conference on Java Grande.

[17] Nguyen, D. and S. Wong. *Design Patterns for Sorting*. ACM SIGCSE 2001.

[18] Merritt, S. *A Logical Inverted Taxonomy of Sorting Algorithms*. Communications of the ACM, Vol. 28, 1 (Jan 1985), 96-99.