

Assignments for an Objects-First Introductory Computer Science Curriculum

Mathias Ricken
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-3836
mgricken@rice.edu

ABSTRACT

Designing an effective curriculum to teach programming and software engineering to beginning students is challenging. An objects-first course prepares students in an excellent way for the 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 five assignments that can be used as teaching tools in an objects-first course to evaluate and reinforce the students' understanding.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming.

General Terms

Design

Keywords

CS1/CS2, objects-first, design patterns, pedagogy, assignments, software quality.

1. INTRODUCTION

From the beginnings of computer science as a discipline, the structure of the introductory curriculum and the role of programming have been debated. Traditionally, programming is taught in an imperative-first style that focuses on programming language elements such as expressions, control structures, and functions.

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 program design. Because of the focus on syntax, imperative-first introductory courses frequently have to use overly simple assignments that do not motivate students, misrepresent the nature of software engineering, and provide students with too little practice of design, testing, and documentation.

An alternative objects-first introductory curriculum that combines

an early-on focus on object-oriented (OO) design and programming with UML diagrams, design patterns, documentation, and unit testing can alleviate many of the concerns voiced in the report. 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 quality factors.

Section 2 discusses some of these quality factors and how an OO approach can improve them. Section 3 describes five assignments that can be used at different stages in objects-first CS1/CS2 courses to refine the students' understanding of OO principles and practices. Section 4 makes an attempt to evaluate the objects-first approach.

2. SOFTWARE QUALITY FACTORS

Software needs to exhibit certain characteristics, which can be external and 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 design, implementation, and extension of the product. Flexibility, extensibility, testability, and documentation fall into this group. In the end, 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, and foresighted 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 program's overall quality. The effects of object orientation on several 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 that extend the superclass. This separation of variants and invariants 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. In contrast to correctness, which derives from system behavior in states covered by the specifications, robustness deals with unexpected situations.

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, it can be extended by changing the variants without undermining the structure enforced by invariant code. Again, the correct decomposition of a problem into variants and invariants is necessary to find the proper abstractions.

2.3. Flexibility and Extensibility

Flexibility and extensibility are two of the internal quality factors that are important in software development. 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, testing and documentation becoming more important as the projects size grows. Unfortunately, internal documentation is often neglected both in computer science 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 [2] 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 regression testing. The use of a unit testing framework like JUnit [3] provides a formal way to ensure that a change has no negative effects on the system. The 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 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 even *requires* paying attention to these factors.

3. SUITABLE ASSIGNMENTS

As the previous section has described, OO concepts can improve the quality factors important to software development. Unfortunately, traditional imperative-first introductory curricula typically do not teach object orientation until the end of the first semester and do not place much emphasis on it. In most cases, departments provide an upper-level course on "OO methodologies" instead. 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. A typical toy example that uses an abstract animal, able to make a sound, and concrete animals such as dogs and cats, simply is not a compelling example of how OO design and programming can help. To demonstrate the importance of secondary or internal quality factors such as robustness, flexibility, and extensibility, exercises must go beyond small-scale programs.

Finding suitable assignments for an objects-first 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 five sufficiently complex yet entertaining assignments that can be used in objects-first introductory courses. The Shape Calculator is used in our institution's objects-first CS1 course; the remaining assignments are used in CS2 courses.

It is important that the assignments described below are given in the context of a comprehensive instruction in OO design and programming. For all of them, students should possess a solid

understanding of inheritance, composition, delegation model programming, polymorphism, closures, and anonymous inner classes. Most of the assignments also require the use of several design patterns, such as abstract factory, composite, singleton, state, strategy, visitor, and decorator [2]. Since all of the assignments involve GUI programming, the model-view-controller (MVC) pattern is also used extensively.

3.1. Shape Calculator

The Shape Calculator [5] is a GUI application that can compute the area of an arbitrary shape. The class representing this shape can be compiled later and loaded into the program at runtime. At our institution, the project is currently being assigned as a 1.5 hour laboratory exercise in the 9th week of the CS1 course that extends into a week-long homework assignment, which takes students 3 to 6 hours to complete.

The project is used as an introduction to GUI programming in Java. Creating simple window frames, adding GUI components, and drawing on the window panel are valuable skills by themselves that will be used in most future assignments in CS1/CS2. Here, however, the project is also being used to dramatically visualize the compelling use of OO principles like encapsulation, inheritance, and polymorphism, and how they make an application extensible.

The visual nature of the project mixes fun with theory and motivates students to learn more about OO concepts. The project is relatively small – the sample solution is only about 1,000 lines of code – yet is able to demonstrate powerful capabilities that would be nearly impossible to produce without using an OO design. To further narrow down the size of the project, the assignment is set up in four parts and provides detailed step-by-step instructions in the beginning, while giving the students more freedom and responsibilities towards its end.

By designing a completely functional, non-trivial GUI program from scratch, students learn to use Java's GUI framework and the use of anonymous inner classes as event listeners. Anonymous inner classes are also used inside abstract factories, which enable the program to manufacture instances whose concrete class is unknown to the application. This reinforces the concepts of closures and the abstract factory pattern.

At the time this assignment is used, students should be acquainted with the abstract factory, composite, and singleton patterns. Beyond these requirements, there are two difficulties which instructors need to consider: Often, students have problems understanding the service-oriented nature of screen painting. It is not immediately obvious to students that control flow is actually inverted and that the GUI invokes their code. Students will also need to learn that local variables must be declared final if they wish to access them from within an anonymous inner class, which also means the variables' values cannot be changed. Good use of direct access to variables outside the anonymous inner class but within its scope is necessary to fully utilize inner classes.

3.2. Tournament Tree

In the Tournament Tree [6] exercise, students design and implement a GUI application that allows the user to edit, display, and simulate a tournament tree as used in a single-elimination sports tournament. The project is designed for students in a curriculum that has included some OO implementations of common data structures and provides an interesting application of binary trees in a sports context that is familiar to students.

The assignment introduces “process flow diagramming”, an application of the delegation model, which in combination with an OO data structure, the visitor pattern, and anonymous inner classes converts the design of complex algorithms into a systematic process. This technique eliminates the need for the large nested if-statements that are usually seen in complex algorithms and replaces them with nested visitors delegating calls from one object to the next. These calls employ polymorphism to automatically perform the correct behavior based on the system state. Using nested visitors in place of if-statements is less error-prone, since the programmer is forced to consider all possible states in which the system can be. It is impossible to accidentally ignore a condition that is rare or unexpected.

To be prepared for this assignment, the course needs to have covered using an OO binary tree structure and implementing algorithms operating on it using the visitor pattern. Instructors need to make sure students do not attempt to solve the problems of manipulating the binary tree using a global approach – the result would be a complicated, imperatively written solution. Instead, students should focus on local decisions and make use of delegation and recursion to limit the scope and complexity of the problem. By closely following the process flow diagrams, the task can be transformed to a number of local decisions that easily map to nested visitors and thus to simpler and more robust code.

3.3. Sorting

In the Sorting assignment [6], students experiment with bubble sort, merge sort, quick sort, and several other comparison-based algorithms and display their effects through animation.

All sorting algorithms are modeled using an OO formulation of Merritt's divide-and-conquer taxonomy [7]: The array is split, the subarrays are recursively sorted and then merged again. Since this general process is invariant, it can be encoded in a superclass employing the template method pattern [2]. The differences between the algorithms are relegated into variant subclass code, and the decorator pattern is used to augment the sorting algorithms with animation capabilities [8]. The assignment serves as an example of how a separation of variants from invariants makes a program more abstract and easier to understand, and the addition of animation in an unobtrusive way stresses the extensibility of OO solutions.

The use of Merritt's taxonomy and OO design enables our students to compare and contrast the different sorting algorithms with much attention to detail. The division of an algorithm into “split”, “recur”, and “join” phases often simplifies the memorization of the algorithms and the discussion of their asymptotic behaviors.

We also use this assignment to introduce several elements of procedural programming, such as arrays, loops, and conditionals, whose presentation we delayed in favor of OO concepts. An imperative-first course could take the opposite direction and adapt this assignment to present inheritance, polymorphism, and several design patterns to students trained in procedural programming. Like the other assignments, this project offers the necessary complexity to highlight the positive aspects of object orientation, and the application of both procedural and OO concepts makes it an ideal candidate for the introduction of the respective other paradigm.

3.4. Games for Two

The Games for Two assignment [6] presents students with a framework for playing two-person turn-based board games. 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 [2] 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 reuse, 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. The game model alternately 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.

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.

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.

3.5. Marine Biology Simulation

The Marine Biology Simulation [9] is derived from the Java AP Marine Biology Simulation Case Study [10] used in high school AP computer science courses. 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 reimplement 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 [2] 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.

The visual way of presenting these subjects enables students to reinforce the concepts learned so far in a compelling and immediate manner. 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 anyway.

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 reimplementing 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 naturally presupposes students have been taught the basic OO skills mentioned in Section 3, as well as additional design patterns, especially command and factory method [2]. 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. We also received questions about the callback-style communication between a fish and its environment.

4. AN EVALUATION OF OBJECTS-FIRST

In an attempt to study the quality of education students in our objects-first CS1/CS2 courses receive, we compared their performance in a systems class to that of graduate students in the same class. 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 any hypotheses. 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 [11], however, suggest the opposite. The main predictor for success in assignments, exams, and courses in general 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 futures indubitably 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 [12].

The size and nature of the assignments used in our introductory courses enable students to effectively participate in upper-level courses as well as in the development of complicated large-scale software. In our production programming course, students have employed OO design and unit testing to develop, maintain, and enhance *DrJava* [13], an integrated development environment of considerable complexity. This enterprise has 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 [14].

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. 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 shows that an objects-first introductory curriculum provides a superb alternative to the prevalent imperative-first approaches. For such a curriculum to be successful, though, compelling examples and assignments of adequate complexity are required. The five 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 end is an introductory curriculum that addresses several of the problems mentioned in the ACM/IEEE-CS report and that provides students with the skills to write better software.

6. REFERENCES

- [1] *Final Report of the Joint AMC/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science*, ACM/IEEE, 2001.
- [2] <http://java.sun.com/j2se/javadoc/>
- [3] <http://www.junit.org/>
- [4] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [5] Cheng, E., D. Nguyen, M. Ricken, and S. Wong, *Nifty Assignments: Abstract Factories and the Shape Calculator*. ACM OOPSLA Educators’ Symposium 2004.
- [6] <http://www.owlnet.rice.edu/~comp212/04-spring/assign.shtml>
- [7] Merritt, S. *A Logical Inverted Taxonomy of Sorting Algorithms*. Communications of the ACM, Vol. 28, 1 (Jan 1985), 96-99.
- [8] Nguyen, D. and S. Wong, *Design Patterns for Sorting*. ACM SIGCSE 2001.
- [9] Cheng, E., D. Nguyen, M. Ricken, and S. Wong, *Nifty Assignments: Marine Biology Simulation*. ACM OOPSLA Educators’ Symposium 2004.
- [10] The College Board, *Java Marine Biology Simulation Case Study*, <http://apcentral.collegeboard.com/>
- [11] Ventura, P., *On the Origin of Programmers: Identifying Predictors of Success for an Objects-First CS1*, Dissertation, SUNY Buffalo, 2003.
- [12] Alphonse, C. and P. Ventura, *Object Orientation in CS1-CS2 by Design*, ACM ITiCSE 2002.
- [13] <http://drjava.org/>
- [14] Allen, E., R. Cartwright, and C. Reis, *Production Programming in the Classroom*. ACM SIGCSE 2003