

Position paper for OOPSLA 2001
Fifth Workshop on Pedagogies and Tools for Assimilating Object-Oriented Concepts

OOP in Introductory CS: Better Students Through Abstraction

Dung Nguyen and Stephen Wong
Department of Computer Science
Rice University
6100 Main Street, MS 132
Houston, TX 77005-1892
dxnguyen@rice.edu, swong@rice.edu

Abstract:

Abstract thinking is difficult for many students to learn, but is a crucial component for learning computer science. We developed and implemented an OOP-first that not only teaches fundamental CS principles and abstract thinking but also inspires and motivates students to push the boundaries of their abilities. This process utilizes design patterns as a "meta-language" for discussing and emphasizing abstract concepts. This is combined with carefully constructed lecture and laboratory materials that leverage intrinsic understandings of object behavior. Students learn the power of abstractly decomposing a problem and expressing it in their designs. We exploit state-of-the-art development tools in the laboratories to help the students maintain a proper focus on the abstractions. We designed powerful and appealing programming assignments to inspire and motivate while driving home fundamental CS principles. The careful integration of many pedagogical facets results in students who possess strong abstraction skills and who can produce powerful, robust, flexible and extensible software.

Position:

The CS pedagogy field struggles with how to introduce OO concepts at the introductory level. Conferences abound with arguments of whether to present procedural first or OO first, specialized software teaching environments and tools, and discussions as to why OO is or is not too difficult for introductory students. Unfortunately, myths about the advanced nature of OO concepts hamper progress in CS pedagogy. We believe that when the role of OOP/OOD is carefully analyzed with respect to the goals of introductory CS, it transforms from an advanced subject to a tool for elucidating fundamental topics.

At Oberlin College, where Stephen Wong taught up through last spring, the commonly used approach of mixing OO ideas into a fundamentally procedural training was used up until 2 years ago. Even though Java was used, this method was fraught with problems. The students ended up knowing syntax but not programming. They had no experience in design or large-scale projects. The conceptual jump to functional programming in the 3rd semester Scheme course was difficult and material seemingly unrelated to their introductory experiences. And the course was less than inspiring as the students were not satisfied by the typically small and simplistic programs they coded.

To remedy these problems we developed a radical "objects-first-with-design-patterns" introductory sequence that leveraged the students' intrinsic understanding of object interactions and behaviors, which Stephen Wong implemented at Oberlin over the last two years. CS at the introductory level is more than just learning specific programming notions however. One of its fundamental goals is to train abstract thinking, the basis of the science of computing. Unfortunately, this is also at the core of the difficulty in introductory CS pedagogy. Students have great difficulty in changing their thought paradigms from a typically very concrete specifics-oriented approach to an abstract modeling approach. Our new methodology thus uses OOP as a tool to teach students to abstractly decompose problems and express their ideas in terms of abstract object models. This provides a strong framework in which fundamental CS notions are taught in a language and paradigm independent manner, creating a unified view of the whole field.

Our approach utilizes a carefully constructed roadmap that relates fundamental CS concepts such as abstract behavior, encapsulation, recursion and composition through design patterns such as the strategy, state, and visitor patterns. Design patterns provide very tangible examples of good abstraction plus an industry standard vocabulary with which to discuss those abstractions. This differs from a popular view of design patterns as “recipes” for software solutions. In a nutshell, design patterns work because they are embodiments of fundamental CS principles, which are abstract notions. Design patterns thus enable the students to focus on abstraction by providing a meta-level between concrete objects and difficult abstract concepts.

Let’s take a quick look at several design patterns to see their relationships with pedagogical goals: The strategy design pattern is one of the most basic patterns because it embodies the fundamental notions of abstracted behavior implemented through polymorphism. Students using a strategy pattern learn to focus on invariant, abstract system behavior without getting bogged down in specific variant implementations. When analyzing a system, it is very easy for students to be drawn to the concrete specifics of the current problem, ignoring the generalizable aspects of the situation. A simple reminder that the strategy pattern is applicable in a particular situation immediately changes the students’ perspective to a higher, more appropriate abstraction level. The composite pattern reinforces the concepts of data abstraction and opens the doors for the proper “layered” view on recursion. The state and decorator design patterns are very useful for cementing the notions of indirection and dynamically changeable behavior. The state pattern is also crucial in helping students learn to model systems as state machines. The visitor design pattern is a beautiful example of using polymorphic double-dispatching to provide abstracted behaviors to abstracted data structures. It also exemplifies the notion of decoupling variant and invariant behaviors, a central idea in the proper abstraction of a problem. Likewise, other design patterns are used to explicitly target key OO concepts and fundamental software engineering principles. It should be re-emphasized that the students do not study the design patterns as an end unto itself, but rather they *use* them as meta-language and thinking tool with which to abstractly decompose a problem into its key programmatic constituents. Our publications, listed below, illustrate many of the key ideas and techniques used in our approach.

One of key techniques used to achieve the course goals is the tight integration of technology with the pedagogy. Software design tools are used to enable the student to concentrate on the important concepts of the problem and the design of their programs rather than syntax, implementations or API’s. Rapid application development (RAD) tools enable the students to concentrate on the interactions between objects, particularly the very tangible GUI components, without worrying about the more advanced coding techniques used to implement them. Later in the semester, the students learn the techniques to create GUIs by hand, when they have progressed to the point where they can handle the nuances and complexities involved. This enabled us to create laboratories that are fun, interesting and result in full-sized programs that are enjoyable just to play. Plus, with the larger program development capability, group projects to teach team skills could be designed that truly demonstrated the issues involved with large software projects.

Does all this come for free? Of course not. Some topics, such as formal loop semantics and detailed complexity analysis are delayed until later in the curriculum. (Simple complexity analysis is covered though.) Likewise specialized and optimized coding techniques for graphics and data collections processing are left for more advanced courses. This is not an issue of simply “not enough room” however. It’s really a question of the proper location of these topics in the overall curriculum. One of the basic ideologies here is that it is best to teach introductory students fundamental driving principles of computer science such as abstraction, encapsulation, recursion, composition, and delegation, without worrying about specific language-oriented issues. We feel that students best handle those issues once they have enough experience and perspective to truly appreciate the subtle issues involved and to comprehend the engineering tradeoffs that need to be weighed. It can also be expected that the students may have less total experience in writing traditional procedural type algorithms, though the students are very comfortable and practiced with recursive algorithms. On the other hand, they will be much more advanced in designing flexible, robust and extensible programs. Their procedural skills are honed in upper division courses that concentrate on applicable areas such as operating systems and graphics where highly optimized algorithms are needed.

Group projects serve two purposes: one is to teach team development skills, the other to provide a framework for students to both demonstrate and integrate their new OO skills. To get an idea of how far students can progress with in such a course, we need only look at the students' accomplishments in their final group projects. The group projects not only bring together all the concepts and techniques used in the course, but also play a very large role in teaching the students how to work in design and programming teams. At the end of the first semester, the final group project is typically to write the arcade game "Frogger". This application requires a full GUI, animations, multiple threads, and scoring capability. At the end of the second semester, the final group project is a networked application of the students' choosing. Last year's projects included remote distributed agents with its associated peer-to-peer framework, and a client-server multi-user game. These applications, as well as using all the technologies of their first semester projects, also included remote method invocation (RMI) techniques, dynamic remote code loading for true object transmission, lambda calculus implemented via inner classes, handshaking protocols, and model-view-controller architectures. Many of the techniques used were not covered in class but the undaunted students dove right in and figured out solutions for themselves. The level of student involvement and excitement was truly gratifying.

It took a tremendous amount of preparation, and most importantly, rethinking of traditional CS materials to develop our "objects-first-with-design-patterns" approach. But in the end we believe that the synergy between the focus on abstraction, fundamental CS principles, design patterns and technology results in an effective, proven pedagogy where the students hone abstractive skills while creating programs of unprecedented complexity and capability. The students demonstrate software with flexible, extensible and robust designs. Their programs display full graphical user interfaces, animations, dynamically modifiable behaviors, and networking capabilities. Team programming skills are practiced on projects with enough depth and complexity to truly simulate real-world scenarios. And just as importantly, the students have fun creating programs that they feel are relevant, exciting and cutting edge.

References:

- D. Nguyen and S. Wong, "Design Patterns for Sorting," SIGCSE Bulletin **33**:1, March 2001, 263-267.
- D. Nguyen and S. Wong, "Design Patterns for Lazy Evaluation," SIGCSE Bulletin **32**:1, March 2000, 21-25.
- D. Nguyen and S. Wong, "Design Patterns for Decoupling Data Structures and Algorithms," SIGCSE Bulletin **31**:1, March 1999, 87-91.
- D. Nguyen, "Design Patterns for Data Structures," SIGCSE Bulletin **30**:1, March 1998, 336-340.

Biography:

Dung Nguyen received his Ph.D. in Mathematics from the UC Berkeley in 1981. His first formal training in Computer Science was with the Institute For Retraining In Computer Science (IFRICS) at Kent State in 1985 and 1986. To further expand his knowledge, he enrolled in many Computer Sciences courses offered at Stanford, UC Berkeley, and UCLA on various subjects ranging from computer graphics, compiler construction, to CORBA programming and network programming. He also participated in various NSF sponsored workshops on object-oriented programming (Illinois State), artificial intelligence (Temple University), and computer networks (University of Dayton). Before joining Rice, he was a visiting faculty at Pepperdine University. There, he and Stan Warford co-crafted a brand-new Computer Science/Mathematics curriculum with a strong emphasis on object-orientation.

Stephen Wong received his Ph.D. in Physics from MIT in 1989 where he was a Howard Hughes Doctoral Fellow. After a stint at the Hughes Research Laboratories as a Member of Technical Staff, he started his teaching career at California Lutheran University. There, he was the only member of the combined Math/Physics/Computer Science Department to teach in all three disciplines. During this time, he and Dung Nguyen formed a consultant team for Kodak, developing object-oriented scientific analysis software. He then moved to Oberlin College where he transitioned to teaching computer science full time. He just moved to Rice University this summer to work closer with Dung Nguyen on an introductory CS textbook based on their work both at Oberlin and at Rice.