An Aspect-Oriented Approach to the Undergraduate Programming Language Curriculum

Mark A. Sheldon and Franklyn Turbak
Wellesley Collge

Abstract

There are two key forces impinging on the modern CS curriculum: an increasing desire for new topics/courses is squeezing out existing ones; and project-based course work has led to a notion of *just-in-time* teaching in which particular skills are not bundled into a particular course, but are covered at a point when they are needed for particular project work.

Programming languages courses are feeling the pressure of the first force: as a body of knowledge, they are not seen as immediately relevant, and such courses are often not required and in some cases are even being phased out. Yet, programming languages as a discipline contains many ideas that any well-educated computer scientist needs to know and which are relevant to a wide variety of projects, so these concepts can still be taught, albeit in a more piecemeal way.

There seems to be a movement towards a system in which information from many standard courses is going to be distributed across the curriculum. In the face of these changes, it is worth viewing the CS curriculum not only as a collection of courses, but as a set of *aspects* that can be combined in different arrangements to produce various projects/courses.

In this paper, we consider the benefits and challenges of an aspect-oriented curriculum, particularly in regard to its impact on topics in programming languages.

The Changing Computer Science Curriculum

There is a revolution brewing in the undergraduate Computer Science curriculum. The traditional model of standard courses arranged in a prerequisite tree/DAG is showing signs of strain. Introductory courses, which are often programming-intensive, can give students the mistaken impression that CS = programming, discouraging students who would be drawn to other aspects of CS [BF05]. A typical course includes topics that students will not apply until much later, if ever, in their coursework and perhaps even their careers. There are often long prerequisite chains that prevent a student, especially one not majoring in CS, from taking courses that interest them. In a time of declining enrollments, CS departments cannot afford to have courses or topics that are perceived as irrelevant or that serve as a barrier to students interested in a particular subdiscipline of CS (e.g. artificial intelligence, bioinformatics, data mining, human-computer interaction, graphics, robotics, web search, etc.)

Given the interdisciplinary nature of CS, it is important to design the CS curriculum to be accessible to a broad audience. This is particularly important in a liberal arts environment like Wellesley's, where aspects of computer science are relevant to students in interdisciplinary programs like media arts and sciences, cognitive science, neuroscience, computational biology, and computational chemistry, as well as general sciences and economics. One way we have addressed this at Wellesley is by shortening prerequisite chains. Students need only one prerequisite to take *Computer Vision* and *Multimedia Design and Programming*, and two prerequisites to take *Artificial Intelligence*, *Computer Graphics*, and *Databases with Web Interfaces*.

Courses need to show students that CS transcends programming and involves aspects of problem solving, design, teamwork, and communication with users. For these reasons, many computer science courses are now incorporating open-ended projects. Some institutions, such as MIT and Olin College, have scrapped the traditional curriculum and are experimenting with approaches in which introductory and more advanced courses are interdisciplinary and project-driven. Such courses often integrate topics that would be taught in several different traditional courses. For example, an introductory robotics course might teach concurrent programming, electronics, planning, signal processing, and control system theory in the context of solving a particular problem. These are taught as needed in a just-in-time way.

While they address many problems with the standard curriculum, shortened prerequisite chains and project-based courses have problems. Covering only what is needed for the current project can give a shallow level of knowledge — an appetizer for the subject matter, but not the entree. Since different courses may need the same just-in-time material, there is the threat of duplication. For example, at Wellesley, C/C++ programming is independently taught in *Systems Programming*, *Computer Graphics*, and *Computer Security*, because they share no prerequisite course teaching C. And focusing on a project can squeeze other material out of a course. When we adopted a final project in our *Data Structures* course, for example, our beefed-up coverage of GUIs necessary for the project forced us to eliminate material on tree and graph algorithms. This creates pressure to move material elsewhere in the curriculum.

New courses in the curriculum are another source of pressure on traditional courses. At Wellesley, we have recently added courses on *Bioinformatics*, *Computer Security*, *Human-Computer Interaction*, and *Web Search & Data Mining*. Given our decreasing enrollments, these new offerings have forced us to teach other courses less often or cancel them altogether.

These developments have had a major effect on the programming languages curriculum. At Wellesley, our *Compilers* course hasn't been taught in five years. There are pressures to change the status of our *Programming Languages* course, which is currently a requirement for the major, to an elective.

We strongly believe that there are fundamental concepts in these courses that any educated computer scientist needs to have in their intellectual tool kit.

An Aspect-Oriented Curriculum

In the *aspect-oriented programming paradigm* [KLM97], programs can be described in terms of *aspects* that cut across traditional program organization boundaries. By analogy, this paradigm provides a model for viewing the CS curriculum as a collection of aspects that cut across course boundaries. In this model, concepts traditionally covered in a single course can instead be distributed across multiple courses: Even though a curriculum might not have a *Programming Languages* course, say, it may include *Programming Languages* aspects.

Thinking of a CS curriculum in terms of *aspects* striped across various courses allows us to systematize essential topics, work towards some degree of uniformity, saves duplication of effort in the development of materials and approaches, and allows us to save critical ideas from courses that are no longer taught.

There have always been ideas that cut across the CS curriculum. Ideas like abstraction and modularity apply to every course and project as do skills like debugging and testing. The "knowledge areas" and "performance capabilities" in the 2005 ACM Computing Curriculum [CC05] and "topics" the 2007 LACS Curriculum [LACS07] can be viewed as aspects. Project-based CS courses magnify this effect.

Here are some examples of how topics typically presented in programming languages courses can be taught in a more aspect-oriented way:

Programming paradigms: PL courses often expose students to new programming paradigms. Imperative and object-oriented paradigms are well-covered in the modern curriculum, but function- and logic-oriented paradigms are not. The fact that so many languages are used in today's curriculum[1] means that there is a greater likelihood of exposure to other paradigms. At Wellesley, students see function-oriented programming in Common Lisp in the *Artificial Intelligence* course and in ML in the *Theory of Computation* and *Programming Languages* courses. They are also exposed to function-oriented ideas like compositional programming with immutable lists in Java in the CS1 course. Logic-oriented programming is currently not covered in our curriculum, but could be an aspect added to the *Artificial Intelligence* course.

Metaprogramming/Interpretation: Even if students don't take a *Programming Languages* or *Compilers* course, they can still be exposed to the notion of programs that manipulate other programs. Javascript's `eval()` function makes it easy to write programs that create and run other Javascript programs. Database courses discuss programs that generate and run SQL queries. A web browser for a simple subset of HTML is a nice example of an interpreter.

Scanning and Parsing: These topics are typically covered in an elective *Compilers* course that many students will not take. We have incorporated them into our required *Theory of Computation* course, where it fits nicely with automata/grammar theory. The additional material requires an additional class meeting each week.

Type Systems: These can be studied in the context of a variety of programming languages. Java's generics facility makes some advanced ideas (universal polymorphism and bounded quantification) easier to discuss outside a *Programming Languages* class.

Theorem proving: It is natural to include exercises with theorem provers in a algorithms course or theory of computation course, but these are already rather packed. Another possibility is a discrete math course in which proof ideas are first covered.

Concurrency: This is a particularly pervasive topic. Concurren¡cy and threads are covered (rather differently) in our *Robotics*,¡ *Systems Programming*, and *Web Databases* courses. Concurrency also arises naturally in GUI/HCI-based projects. Lynn Stein at Olin even teaches concurrent programming in a CS1 course.

Many other areas of CS are amenable to aspect-oriented presentation: algorithms and artificial intelligence (presented as-needed), simulation/modeling and software engineering (used in the context of projects). On

[1]The 16 Wellesley courses with substantial programming use 16 different languages (some of these courses share the same language while some use more than one).

the other hand, other areas seem resistant to the aspect-oriented approach. For example, it's hard to imagine teaching a hardware, networks, or operating systems in chunks distributed across several courses.

Challenges

Despite the advantages of an aspect-oriented curriculum (greater relevance, reinforcement of the connections between major ideas, shorter prerequisite chains, ability to include key units from courses no longer taught), there are significant challenges.

It is difficult to modularize the curriculum this way. The most apparent issue is duplication of effort. There is overhead in teaching a concept, such as a sorting algorithm, interpreter structure, or environment model. Paying this cost multiple times decreases the time available for other topics (and risks boring students who have seen the material one or more times before).

The aspect-oriented model will only make this situation worse. Particularly if the prerequisite structure is reduced or removed, a topic will come up (eg, mutexes and condition variables), and the instructor can make no assumptions about the background of the students in the class. "Memoizing" such units by offering self-paced units or mini-courses is one way to address this problem, but it imposes scheduling constraints, since some members of a team may need one or two weeks of just-in-time, specialized instruction.

Prerequisites become harder to think about in this model and make some topics resist modularization. Certain topics require substantial background. For example, many courses, such as PL, use substantial amounts of specialized notation and vocabulary. Imagine trying to use a denotational semantics without the lambda calculus, a theory of recursion without a notion of fixed points. As another example, a certain amount of set theory, discrete math, and logic may be important if a project bumps up against the limits of computability: Consider a project that involves a mobile scripting application (something like SQL code or a reservation system). Devising a static analysis system for the language to prove that it does not violate security constraints would be a good idea, but it is difficult to teach about such a mechanism if you cannot assume students have a solid (or even common) background.

A related issue arises when one attempts to integrate such courses into an existing curriculum. For example, at Wellesley, we have three different introductory courses intended for different audiences: a CS1 course for prospective majors a web development course for non-majors and a programming and data modeling course for students in the sciences. Each course makes a different language choice: Java, HTML/CSS/JavaScript, and MatLab, respectively. At present, only students who take the CS1 course can go on to the data structures course because of the varying topic coverage among the course and because the data structures course is taught in Java and assumes a certain level of language-specific experience.

The aspect-oriented model requires greater collaboration among faculty members. Courses are often "owned" by one or two faculty members who may be resistant to modifying the structure of a course that works well in order to incorporate new aspects. A faculty member in charge of an aspect may fail to appreciate or support key pedagogical concerns of a project that uses the aspect. The aspect-oriented model is also rather sensitive to the teaching preferences of instructors. For example, in Wellesley's data structures course, some instructors relate Java inner classes to higher-order functions, while others omit inner classes entirely.

The material students will see is rather dependent on the particular path they take through the curriculum. Some valuable material may be lost to some or all students. We fear this would be true of many more abstract or theoretical concepts.

Finally, the question of what a CS degree means becomes rather complex. If there are no standard, required courses (or fewer required courses), and there is a greater variety of student paths through the curriculum, what guarantees can one make about the knowledge of a graduate from the program? Perhaps the path can be tracked and evaluated according to the number and type of aspects visited.

References

[BF05] Lenore Blum & Carol Frieze. "In a More Balanced Computer-Science Environment, Similarity is the Difference and Computer Science is the Winner." *Computing Research News* 17(3). May 2005.

[CC05] Russell Shackelford, et al. "Computing Curricula 2005: The Overview Report." *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education.* Association for Computing Machinery. September 2005.

[KLM97] Gregor Kiczales, et al. "Aspect-Oriented Programming." *Proceedings of the European Conference on Object-Oriented Programming,* Mehmet Akşit and Satoshi Matsuoka, eds. Pages 220–242. 1997.

[LACS07] Liberal Arts Computer Science Consortium. "A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science." *Journal on Educational Resources in Computing (JERIC)* 7(2): 2. June 2007.