

An Aspect-Oriented Approach to the Undergraduate Programming Language Curriculum

Mark A. Sheldon

Wellesley College
msheldon@wellesley.edu

Franklyn Turbak

Wellesley College
fturbak@wellesley.edu

Abstract

Three key forces are shaping the modern Computer Science (CS) curriculum: (1) new topics/courses are squeezing out existing ones; (2) a focus on “big picture” and interdisciplinary aspects of CS is leading to curricula in which the traditional core courses + electives model is being superseded by a more flexible approach based on tracks/threads; and (3) project-based courses are increasingly relying on a notion of *just-in-time* teaching in which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.

The undergraduate programming language curriculum is feeling the pressure of these forces. Core courses on programming languages and compilers are being changed to electives, relegated to a software/systems track, or phased out altogether. Particular programming languages and programming language concepts are being taught in a more piecemeal fashion on an as-needed basis. Unfortunately, these changes make it increasingly likely that CS majors can graduate without being exposed to certain “big ideas” of programming languages.

Programming languages (along with many other subdisciplines of CS) contains ideas that every well-educated computer scientist needs to know and which are relevant to a wide variety of projects/careers. In a CS curriculum based on tracks and projects, how can we ensure that vital intellectual components are not lost in the shuffle?

One way to think about this problem is to view the CS curriculum as a collection 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 view of the CS curriculum, particularly in regard to programming language concepts.

1. The Changing CS Curriculum

There is a revolution brewing in the undergraduate Computer Science (CS) curriculum. The traditional model of core courses arranged in a prerequisite tree/DAG with electives at the leaves is showing signs of strain. Introductory courses, which often focus on programming, can give students the mistaken impression that CS = programming, discouraging students who would be drawn to other aspects of CS (BF05). A typical core 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 programs cannot afford to have courses or topics that are perceived as irrelevant or that serve as a barrier to students interested in particular subdisciplines of CS, particularly those with an interdisciplinary flavor (e.g., artificial intelligence, assistive technologies, bioinformatics, electronic commerce, graphics/visualization, human-computer interaction, privacy/security, robotics, etc.) CS departments are moving away from the view that CS is a discipline to be studied for its own sake and embracing a view in which CS is a critical component in a broad range of careers in a globally competitive economy.

The rigidity of a CS curriculum based on core courses + electives is giving way to more flexible approaches in which there are many different paths to a CS major. For example, Georgia Tech has adopted a model in which CS students design a major by choosing two of eight *threads*: Computational Modeling, Embodiment, Foundations, Information Internetworks, Intelligence, Media, People, and Platforms (FIG07). The new CS curriculum at Stanford replaces a large collection of core courses by a smaller core and a choice of one of several *tracks*: Artificial Intelligence, Theory, Systems, Human-Computer Interaction, Graphics, Information, Biocomputation, and Unspecialized (Sah08).

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 CS are rel-

evant to students in interdisciplinary programs like media arts and sciences, cognitive science, neuroscience, computational biology, and computational chemistry, as well as economics and other sciences. One way we have addressed this at Wellesley is by shortening prerequisite chains. Students need only one prerequisite to take *Computer Vision* or *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 CS courses are now incorporating open-ended projects. Some institutions, such as MIT and Olin College, have scrapped the traditional introductory curriculum and are experimenting with approaches in which introductory (and also 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.

The just-in-time approach to teaching certain topics used in such courses has some important advantages. Foremost is relevance. Students are much more likely to be involved in a topic when it has an obvious relationship to some problem they care about. Second, an approach that allows students to learn a topic when they need to know it allows for shorter prerequisite chains. This, in turn, allows for students with varying backgrounds and interests, even students from other majors, to get exposed to important topics of interest to them without having to take three or four other CS courses. This increases diversity in the department, addresses issues of low enrollments, and may improve the educational experience for students in general. Finally, such courses expose students to synergies among CS topics and between CS and other fields.

While they address many problems with the standard curriculum, threads/tracks, shortened prerequisite chains, and project-based courses have their own problems:

- In a track-based curriculum, students are no longer necessarily exposed to key ideas from traditional core courses in the tracks they do not take.
- With shortened prerequisite chains, students don't have the full background traditionally assumed for upper-level courses. This means that certain topics cannot be covered in as much depth or that tutorials must be given to bring students up to speed.
- Covering only those ideas needed for a particular project can give a shallow level of knowledge — an appetizer for the subject matter, but not the main course.

- 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.
- Focusing on projects 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 some 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.

2. W(h)ither Programming Languages?

The developments described above have had a major effect on the undergraduate programming languages curriculum. At Wellesley, our *Compilers* elective course hasn't been taught in five years due to a combination of low enrollments and the fact that students prefer electives on "hotter" topics. Although our *Programming Languages* course remains a core requirement, there is strong pressure to make it an elective to give students more flexibility in the major.

At MIT, an overhaul of the curriculum has led to the elimination of the venerable *Structure and Interpretation of Computer Programs (6.001)* course. It is unclear where (if at all) certain topics from this course will be taught in the new curriculum — e.g., higher-order functions, lazy data, interpretation, nondeterministic programming, and logic programming. In a curriculum based on threads/tracks, many students will never encounter these topics and many other big ideas from programming languages unless they follow a thread/track that includes some course on this discipline.

One reaction to this state of affairs is acceptance. We can accept that many CS majors will not be exposed to the big ideas from programming languages. Instead, we can focus on the topics that will be covered in the courses/tracks that specifically teach programming language concepts.

However, we have a different reaction. We strongly believe that there are fundamental concepts in programming languages (as well as other subdisciplines of CS) that any well-educated computer scientist needs to have in their intellectual tool kit. So the question we have is:

Can anything be done to ensure (or at least increase the probability) that most CS students are exposed to the big ideas of programming languages?

Of course, exactly what ideas in programming languages are sufficiently “big” is open to debate. (Below, we give some concrete examples of what we think should be included). But, for the sake of argument, let’s say that we are given a list of such ideas. Our goal is to figure out ways to embed these ideas in the new curricular landscape in such a way that most paths through this landscape will visit the ideas.

3. 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 (or thread/track) might not have a *Programming Languages* course, say, it might 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, reduce duplication of effort in the development of materials and approaches, and preserve 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” in the 2007 LACS Curriculum (LACS07) can be viewed as aspects, although they tend to be organized into traditional courses rather than cutting across several courses.

We now give some examples of how some essential ideas from programming languages can be taught in a more aspect-oriented way:

Metaprogramming: The notion that programs can manipulate other programs (e.g., interpreters, compilers, program analyzers) is a fundamental idea in programming languages — one that students typically find both mind-blowing and confusing. Even if students don’t take a *Programming Languages* or *Compilers* course, we think that it is still essential that they not only be exposed to this idea, but get some hands-on experience with it by building simple interpreters and translators.

There are many opportunities for such activities elsewhere in the CS curriculum. Writing a program for interactively testing a collection of functions/procedures/methods involves a simple interpreter and read/eval/print loop. JavaScript’s `eval()` function makes it easy to write programs that create and run other JavaScript programs. Examples of interpreters for tree-structured languages include an expression evaluator, a web browser for a simple

subset of HTML, and a program for preprocessing and running SQL queries. These sorts of metaprogramming projects may be easier to incorporate into a course if the trees are supplied by a black-box parser. In a course that already covers trees as a topic, such as *Data Structures* or *Algorithms*, the abstract syntax trees used in metaprogramming are a compelling example of trees that the students use every day.

First-class/Higher-order Functions: We believe that first-class/higher-order functions are an essential idea in CS. Not only do they provide important modularity benefits in programming, but they lead to powerful new ways to think about solving problems in any discipline (e.g., a problem solution can take a strategy as an input). Our experience shows that the notion of first-class/higher-order functions is one that does not come naturally and is difficult to “get”, which makes it all the more important that we teach it explicitly. The utility of such functions is easily motivated by Google’s MapReduce framework (CC05), which is a “killer app” illustrating processing that involves functional inputs.

Although first-class/higher-order functions are most naturally introduced in the context of functional programming languages, there are many other contexts in which they can be introduced:

- In Java, anonymous inner classes can be used in many situations that call for higher-order functions, such as callback functions for GUI components or methods that map over collections.
- Since JavaScript has first-class functions, a web design course can illustrate them in the context of event handlers, processes that map functions over array elements, etc. They are also central to object-oriented programming in JavaScript; a method is just an object property that is a function.
- In C, a structure can have a field that is a function pointer, a feature that is used to represent file system objects in the Unix/Linux file system.

Type Systems: Traditionally, topics related to type systems (such as type checking, type reconstruction, type casts, universal polymorphism, and bounded quantification) have been taught only in programming language courses. But the popularity of Java combined with the sophistication of its type system allows many of these topics to be discussed in the context of Java programming. For example, the *generics* system of Java 1.5 involves aspects of universal polymorphism, bounded quantification, and type reconstruction. Java is certainly not the *simplest* language in which to introduce these ideas, but at least it’s a popular language that illustrates their importance in practice.

Of course, such ideas are more naturally discussed in the context of languages like ML or Haskell, which is an added benefit of including such languages *somewhere* in the CS curriculum. There are many other compelling reasons to use these languages — e.g., for illustrating sum-of-product data types and pattern matching in the context of tree programming or metaprogramming.

Concurrency: This is a particularly pervasive topic: almost any substantial project or CS track naturally involve aspects of concurrency. In our curriculum, concurrency and threads are covered (rather differently) in our *Robotics*, *Systems Programming*, and *Databases with Web Interfaces* courses. Concurrency also arises naturally in GUI/HCI-based projects. Lynn Stein argues compellingly that concurrency is a fundamental problem solving technique that belongs in the introductory programming curriculum; she views computation as *interaction* (as opposed to *calculation*) (Ste98).

Scanning and Parsing: These topics are typically covered in an elective *Compilers* course that many students will never take. However, we believe that they are important enough for students to see elsewhere in the curriculum. At Wellesley, we have incorporated them into our required *Theory of Computation* course, where they are used as practical applications illustrating the automata and grammar theory taught in the course. Incorporating this new material in the course required adding a new class meeting each week.

Programming paradigms: Programming language 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 different languages are used in today’s project-oriented curricula¹ means that there are many opportunities for exposing student to new paradigms in other courses.

For example, 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. JavaScript and Python are two popular languages in which function-oriented ideas are natural to explore.

Logic-oriented programming is currently not covered in the Wellesley curriculum, but could be an aspect added to the *Artificial Intelligence* course or *Databases with Web Interfaces* course.

Programming language design dimensions: Using a wide variety of languages throughout the CS curriculum also increases the likelihood that students get some hands-on experience with languages that differ on key dimensions of programming language design. For example, it’s nice for students to be exposed to languages illustrating the following kinds of dimensions:

- first-order vs. higher-order functions;
- dynamic vs. static type checking;
- monomorphic vs. polymorphic types;
- explicit types vs implicit types (type reconstruction);
- manual storage management vs. automatic storage management (garbage collection).

Dimensions like these are often mentioned in a *Programming Languages* course, but getting hands-on experience with languages illustrating different points in the programming language design space is a better way of “getting a feel” for the dimensions. This experience can be enhanced if instructors are willing to explicitly discuss the benefits and drawbacks of languages along certain dimensions, even when they are not being used in a *Programming Languages* course.

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

A particular application for aspects arises when trying to integrate interdisciplinary or specially targeted courses into an existing CS program. 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 uses different languages: 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 courses and because the data structures course is taught in Java and assumes a certain level of language-specific experience.

One way to integrate the courses better into our curriculum would be to specify the prerequisites for the data structures course in terms of aspects (such as conditionals, method/function/procedure definition, recursion, looping, etc.) and then insist that each of the three introductory courses cover these aspects.

4. Challenges

Despite the advantages of an aspect-oriented curriculum (greater relevance, reinforcement of the connections be-

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

tween major ideas, shorter prerequisite chains, ability to include key units from courses no longer taught), there are significant challenges to address.

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, a type system, or an environment-model interpreter. 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).

This is a problem with threads/tracks, but the aspect-oriented model will only make this situation worse. Particularly if the prerequisite structure is reduced or removed, a topic will come up (e.g., 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. MIT’s *Elements of Software Construction (6.005)* course, for example, requires that students unfamiliar with Java complete a tutorial outside of class by a particular time during the term.

Prerequisites become harder to think about in this model and some topics resist modularization in this way. Certain topics require substantial background. For example, many programming language topics use substantial amounts of specialized notation and vocabulary. Imagine trying to use a denotational semantics without the lambda calculus, a type system without deduction rules, or 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.

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 explicitly relate Java’s anonymous inner classes to higher-order functions, while others omit inner classes entirely.

5. Conclusion

In the new, interdisciplinary, thread/track-based curricula, the question of what a CS degree means becomes rather complex. If there are no (or very few) standard, 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?

Aspects give us a way to think about the big ideas of computer science, and programming languages in particular, and to ensure that all or most students who receive a degree in a CS or CS-intensive field possess the key intellectual tools our discipline has to offer.

Acknowledgments

We thank Daniel Jackson for explaining the new MIT EECS curriculum and describing his new *Elements of Software Construction (6.005)* course. We are grateful to Mehran Sahami for summarizing the new Stanford CS curriculum.

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, 2006.
- [CC05] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” *Communications of the ACM*, 51 (1), pp. 107-113. January, 2008.
- [FIG07] Merrick Furst, Charles Isbell, and Mark Guzdial. “Threads: How to Restructure a Computer Science Curriculum for a Flat World.” *SIGCSE ’07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. Association for Computing Machinery, 2007.
- [KLM97] Gregor Kiczales, et al. “Aspect-Oriented Programming.” *Proceedings of the European Conference on Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka, eds. pp. 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.
- [Sah08] Mehran Sahami. Preview of the New Undergraduate Computer Science Curriculum. Slides available at <http://cs.stanford.edu/degrees/undergrad/CurriculumRevision-Preview-04-03-08.pdf>.
- [Ste98] Lynn Stein. “What we’ve Swept under the Rug: Radically Rethinking CS1.” *Computer Science Education* 8 (2): 1998, pp. 118–129.