9-2003

Enhancing the CS Curriculum with with Aspect-Oriented Software Development (AOSD) and Early Experience

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Tzilla Elrad
Illinois Institute of Technology, elrad@iit.edu

Enhancing the CS Curriculum with Aspect-Oriented Software Development (AOSD)

Konstantin Läufer and George K. Thiruvathukal
Department of Computer Science
Loyola University Chicago
6525 N. Sheridan Road
Chicago, IL 60626, USA
{laufer,gkt}@cs.luc.edu

Tzilla Elrad
Department of Computer Science
Illinois Institute of Technology
10 W 31st Street
Chicago, IL 60616, USA
elrad@iit.edu

ABSTRACT

Aspect-oriented software development (AOSD) is evolving as an important step beyond existing software development approaches such as object-oriented development. An aspect is a module that captures a crosscutting concern, behavior that cuts across different units of abstraction in a software application; expressed as a module, such behavior can be enabled and disabled transparently and non-invasively, without changing the application code itself. Increasing industry demand for expertise in AOSD gives rise to the pedagogical challenge of covering this methodology and its foundations in the computer science curriculum. We present our curricular initiative to incorporate a novel course in AOSD in the undergraduate computer science curriculum at the intermediate level. We also discuss recent and planned efforts to integrate coverage of AOSD into existing courses.

Category and Subject Descriptors

K.3.2 Computer and Information Science Education; D.1.5 Object-Oriented Programming; D.3.3 Language Constructs and Features; D.2.3 Coding Tools and Techniques

General Terms

Design, Languages

Keywords

Aspect-oriented software development, AOSD, aspect-oriented programming, AOP, aspect-oriented modeling, separation of concerns, crosscutting concern

1. INTRODUCTION

It is now well-established that *object-oriented software development* (OOSD), which comprises analysis and design methods (OOA and OOD), modeling (UML), design patterns, frameworks, and a smattering of object-oriented languages (C++, Java, and Python, just to name a few), is a significant player in the academic and corporate worlds, with many academic research and commercial applications in one way or another building on the OO paradigm. With the advent of Java in 1995, one could say that an idea whose time came in the 1960s and 1970s with Simula and the 1980s with Smalltalk formally got itself on the map. Objects were truly here to stay.

In some respects, however, it was all too painfully apparent that somewhere along the line, something in the process of educating the world and creating a culture of objects had failed. One of the authors recalls being in the corporate world in the early 1990s as part of an internal task force on object-oriented methods and applying them to a redesign of one of the major product lines. Even then, migrating to object-oriented development was considered a risky venture for a number of reasons. First and foremost, there were few companies in any related business with experience in applying OOP. Second, the learning curve for applying OOP was considered (at best) steep [8]. Third, techniques for reverse engineering existing designs (then and even now) were ad-hoc and fundamentally limited. But more disturbing was that there were few options (professional training or courses at academic institutions in the area) where one could learn how the mysterious phenomenon of OOP actually worked.

With the advent of *software design patterns* [4], a new sense of hope descended upon the software development world, despite not providing a solution to all of the problems. What design patterns provided was the potential to apply OOP without having to bear the burden of coming up with the same designs over and over from scratch. This made it possible to apply OOP using recipes that required little or no modification. In fact, design patterns played a significant role in taming the complexity of OOP and, as a result, made it *easier to teach*.

A similar challenge is being presented today, silently, with the emergence of new (or refined, depending on one's perspective) methodologies for software development, such as *aspect-oriented programming* (AOP) [7]. In AOP, the idea in a nutshell is to capture behavior that cuts across many (or possibly all) units of abstraction in a given software application. Expressed as separate modules called *aspects*, such behavior can be enabled and disabled transparently and non-invasively, requiring no change to the application code itself. Simple examples of aspects are logging and per-

formance analysis. Often these aspects are captured in OO-codes by creating objects to do the actual work and then explicitly using them to achieve the task at hand. To understand the explicit nature of this, it is instructional at this point to download any Java code and search for calls of the form `logger.writeLog("some debug message")` or `System.currentTimeMillis()`. An aspect-oriented way of achieving the same result would be to create an aspect to support the desired behavior implicitly. For example, whenever I enter method `X()`, start the timer. Whenever I leave method `X()`, stop the timer and accumulate the difference from the starting time.

Aspects will present new challenges to software development in the so-called postmodern era. First, thinking in aspects is akin to thinking in objects but a much different thinking altogether at the same time. There are several new twists in the world of aspects: How does one conceive of aspects in the first place? How does one combine aspects to solve problems? What are the pitfalls? One pitfall we see is that composition—an ordinarily safe phenomenon in OOP—is not always meaningful and safe in AOP. Take for example the two aspects we mentioned as examples: performance and logging; they don't mix. As already well known in operating systems and parallel computing research, it is important to remove all debugging code before collecting any kind of performance data. The obvious solution is to disable the offending aspect(s). But herein lies the problem. For something such as logging, the problems are obvious. For other aspects, the problems may be ever so subtle. We will revisit this topic in the general discussion of AOP.

Another important consideration is that aspects do not presuppose an underlying object-oriented environment. Aspect-oriented versions of many languages are already creeping up on the Internet, including C, C++, Java, Python, and others. While this in many ways is similar to the conditions found during the OOP era, it can create a bit of a pedagogical mess. "When should aspects be introduced?" is a question that is harder to answer than the same question with *aspects* replaced by *objects*. With the potential need to learn at least 3 forms of abstraction (procedural, object-oriented, and aspect-oriented), the potential for real confusion exists.

In this paper, while we do not pretend to have the answers for the many questions posed, we describe our attempt to rise to this emerging pedagogical challenge. We do believe that AOP seems to follow OOP most logically and envision our proposed course to be taken after CS2 or a subsequent intermediate course in object-oriented development. At the same time, we see a clear and present need and opportunity to introduce AOP strategically in other areas, such as operating systems, concurrent, and parallel/distributed computing.

2. ABOUT AOP AND AOSD

We begin by defining a couple of terms:

AOP The term aspect-oriented programming is used when referring only to programming (languages mainly), but not the entire life cycle of software engineering.

AOSD The term aspect-oriented software development is an encompassing term that includes AOP as a significant part of its definition. In AOSD, we are referring to aspects in what is considered the gamut of software engineering: requirements, design, implementation, and testing (to name the most significant parts of a *traditional* SE life cycle, known as the *waterfall* model).

The core issue behind AOP is to capture the notion of a *concern* as a modular software unit. By allowing concerns to be modular, the desired goal of having adaptable software is more likely to be realized. AOSD refers to the complete life cycle evolving the aspect orientation process. The terminology is deceptively simple and sometimes confuses the most seasoned computer scientists. An example of a concern is scheduling, which affects all entities in a system that can be run one-at-a-time (such as processes and threads). Processes and threads, on the other hand, are not concerns. They are simply objects that are treated the same way by an operating system, regardless of the scheduling policy used to run them.

Separation of Concerns (SOC) is a core principle of software engineering which has well-established benefits. The introduction of of the OO paradigm addressed part of the challenge of concerns by evolving mechanisms for abstraction, encapsulation, modularity, and hierarchy. Yet, the problem of realizing the SOC principle is at the heart of the ongoing software crisis. Ideally, we would like to have, first and foremost, heuristics to separate concerns and, second, software mechanisms to capture these concerns as modular units throughout the software refinement process. AOSD is an important *evolutionary* step toward achieving the goals of separating concerns.

The object-oriented approach focuses primarily on the data items that are being manipulated; behavior is the secondary focus. The data items are characterized as active entities, because objects perform operations on themselves. Collective behavior is implemented through the interaction of objects. (In early OOP literature on Smalltalk, objects were said to interact via message passing.) Object-oriented methodology works very well for software systems where most of problem domain concerns can be modularized using the dominant decomposition of object orientation. Experience has demonstrated that in more complex systems, especially concurrent and distributed systems, preservation of modularity is not possible for all *concerns*.

Crosscutting Concerns. One special case where a program can fail to preserve modularity is when the implementation of a specific concern crosses the modular representation (or boundaries) of other concerns. A good example of such concern is logging. The concept is similar to what one does when writing a journal—keep a record of the events of significance (or even insignificant ones) for any given day/time. (Many programmers prefer logging to debuggers when it comes to identifying and diagnosing bugs in their programs.) A program code to implement logging may be scattered across different modular components (wherein the various components will share a reference to an object with a dedicated logging). Synchronization, scheduling, fault tolerance and security are extreme examples of crosscutting concerns. Under a dominant decomposition of core functionality, these concerns cannot be modularized. The best one can hope for is to scatter concerns among other program components—components that should ideally not be responsible for addressing the concern.

Weaving. The process of composing core functionality modules with aspects is called weaving. Weaving can be done at

compile time [6] or dynamically at run time [10].

Join Points. A Join Points is defined as a well-defined points in the execution flow of the program. The most prominent example is a method call. A joint point indicates where an aspect can interface with ordinary code. A join-point model provides the common frame of reference to enable the definition of the structure of aspects. Each join point is said to have entry and exit points, which correspond to when a join point is entered or exited (returned from). So if a particular join point is defined as a method call, the logical points of entry and exit are entering and returning from the method, respectively. How this works is language-dependent.

Pointcut. A set of join points. A pointcut expression filters out a subset of join points. This is an important feature of AOP because it provides a concise selection mechanism. A programmer may designate all the join points in a program where (for example) a security code should be invoked. This eliminates the need to refer to each join point explicitly and hence reduces the likelihood that a security code would not be invoked as needed (considering the earlier example, where most capability checks had to be performed explicitly). A Pointcut might also be specified in terms of properties of methods rather than their names.

Advice. Advice declarations are used to define the aspect code that runs at join points. For example, the security-checking functionality described earlier in Java's `SecurityManager` could be run at every join point that is filtered out by an appropriate pointcut expression. For a method call join point there are three kinds of advice found in the literature: *before* advice (advice code runs at the moment before the method begins running), *after* advice (advice code runs at the moment control returns after the method returns), and *around* advice (advice code runs when the join point is reached and it can check for conditions that may control the selection of advice code). Advice is similar to event-handling code and is invoked implicitly.

There are three significant approaches to realize aspect-orientation:

- Extensions to existing languages such as Java, C, and C++ (e.g., Hyper/J [9]).
- Frameworks for introducing aspect orientation without change to the language (e.g., JAC [10]).
- Modeling with (suitable extensions of) UML.

3. PROPOSED AOSD COURSE

In this section, we present our plans for a complete course on AOSD.

3.1 Pedagogical Challenges

It should be clear from the previous section that designing a curriculum around aspects is bound to be a non-trivial challenge. The problems can be summarized as follows:

- Aspect-oriented programming is designed to facilitate dealing with the complex problems associated with code tangling. Mastering the basic vocabulary of AOP is relatively straightforward. However, there are a multitude of implementations and approaches to AOP. Students must be made aware of the choices while not becoming bogged down by trying to understand all of the arcane differences between the various approaches.
- Similar to the problems that accompanied OOP (problems with method resolution in multiple inheritance), a number of issues are likely to emerge as the community begins to *apply* AOP. For example, aspects being applied to pointcuts (selections of join points) can interfere with the performance and predictability of program behavior. (e.g. what happens, say, when *logging* and *performance* concerns are both applied to common pointcuts? The answer would appear to be that the performance concern fails to yield valid performance data; the logging concern affects performance by introducing a non-trivial I/O cost.)

3.2 Prospective Syllabus for the Course

Prerequisite

The envisioned prerequisite is either CS2 (data structures), assuming the CS1-CS2 sequence introduces object orientation from the start and in sufficient depth. Otherwise, the prerequisite would be an intermediate course in object-oriented development taken after CS2.

Textbook

There is presently no undergraduate text book on this topic. We anticipate that one of the outcomes of this initiative will be such a book. Meanwhile, we plan on using the other sources as reference material [1, 2].

Prospective Sequence of Course Modules

A prospective sequence of 3-to-4-week course modules suitable for a 15/16-week semester long course, along with the estimated duration of coverage, is given here. To accommodate a 10/11-week quarter course, one could drop some of the modules or condense the coverage.

Module 1: Programming Languages and Software Engineering Preliminaries (3 weeks)

General paradigms of programming languages (1 week). To motivate the study of models beyond the object model, it pays to arm the student with an overview of different paradigms for computing. Most students today see objects as their first model. Structured (procedural/imperative), functional, logic, dataflow, and very-high-level language paradigms are prevalent, especially in the industry.

Software engineering and quality assurance (1 week). Software engineering has played a significant role in the thinking about how software is developed, especially in defining the notion of a *process*. Knowledge of basic software engineering principles is essential, especially when it comes to large-scale software development efforts. Prospective topics to be covered here are an overview of the various *life cycles*, such as waterfall, spiral, and extreme programming.

Software maintenance and reusability (1 week). In addition to methodology, software engineering methodology introduces a number of *desired properties* when designing a software system. *Ease of maintenance* refers to the ability

to cope with change at a reasonable cost. Another desired property of a software system is *reusability*. Research on frameworks and design patterns addresses the possibilities of reusing existing code and designs.

Module 2: Object-Oriented Programming and Design Patterns (3 weeks)

Object-oriented programming (2 weeks). In order to think about the post-OOP era, it is necessary to understand the object model. The object model, as defined by Booch, is based on four key principles: abstraction, encapsulation, hierarchy, and modularity. Object-oriented programming languages support OOP via specific mechanisms. Polymorphism with late binding is prevalent in all major OOL. Linguistic features such as *abstract classes*, *interfaces*, *delegation*, and *metaprogramming* (or *reflection*) are prominent in most OOL today.

Taming the object paradigm with patterns (1 week). The work on design patterns [4] is actually a tacit recognition of the difficulty of applying OOP successfully. Work on patterns also brought some clarity to what OOP does well and what OOP does poorly. OOP is very good at capturing common attributes and behavior in a vertical fashion. However, when it comes to behavior that affects entities that do not have common *base classes*, it is impossible to address commonality in a horizontal fashion. The linguistic notion of interfaces helps get us part of the way toward the solution but in their own right are ineffective.

Module 3: Aspect-Oriented Programming (4 weeks)

Introduction to aspects (2 weeks). Aspects directly embody the notion of separating concerns. The vocabulary has been introduced in the introduction sections. The same topics will be introduced in this lecture.

The motivation for aspects will be demonstrated with a number of real-world needs. Logging is almost a universal need in today's computing environment. Web logging (not to be confused with weblogs) is relied upon heavily by web application developers and has almost replaced debugging as an effective technique for troubleshooting bugs.

In high-performance computing applications, performance is an aspect that comes up repeatedly. Today's software developers often develop what can best be termed *home-brewed* libraries for gathering and analyzing performance data. Using aspects, it is possible to introduce the capability of performance gathering into any application and remove it when no longer needed without changing the application code itself.

Aspects are often called non-invasive, because they can be introduced into code without altering the code itself. While mostly true, there are potential difficulties (addressed later in the course) when it comes to composition of aspects and unintended consequences. The two aspects presented here make a good example. What happens when the performance and logging aspects interact? The likely answer is that bad performance data is obtained.

Aspect-oriented programming languages (2 weeks). Aspect-oriented programming languages are emerging everywhere. The situation is not much different than when structured and object-oriented programming made their debuts.

Two approaches are presently front runners: aspect-oriented extensions (e.g., Aspect/J, an extension to Java) and aspect-oriented frameworks. Because objects are still considered valuable in their own right, most approaches in one way or another employ an object-oriented language as a significant part of the design.

At least this set would be taught from outside of our own research: Aspect/J, Hyper/J, Demeter/adaptive programming, and Compositional filters.

Module 4: Advanced Topics in Aspects (3 weeks)

Theoretical foundations of aspects (1 week). This will be covering current research on composition mechanisms, semantics, verification, and testing.

Aspects and UML; addressing modeling and design (1 week). The Unified Modeling Language (UML) is widely used in present software engineering practice. We will cover our ongoing research in using and extending UML to accommodate aspects.

Tools support for aspects (1 week). A fairly underdeveloped area at the moment, tools support is likely to evolve by the time we develop this course. The likely scenario is that a number of tools (e.g. Rational Rose and others) will be expanded to address aspects.

Module 5: Applications of Aspects (3 weeks)

Aspects and operating systems. An emerging research area is the use of aspects in operating systems. Operating system design and implementation is one area can make a significant difference, especially when it comes to maintenance and debugging. In addition, aspects are likely to be useful in managing the implementation (code) of an operating system, as well.

Aspects and databases. Aspects are likely to make a difference in databases In current database technology, procedural languages (SQL and several variants thereof) are used to do query processing and integrity *triggers*. Integrity triggers, which often affect the same tables/relations in a database, certainly provide an example of a cross-cutting concern. Given the importance of databases in the industry, we will want to address this topic in the new course (at least peripherally).

Aspects and concurrency. We have existing courses addressing concurrency. Concurrency is one of the places where OOP has not been very successful. The reason is due to a well-documented and established problem known as the *inheritance anomaly*. Using aspects, the problems associated with the inheritance anomaly are potentially eliminated (although this has not been formally proven yet), making it possible to untangle programming libraries that contain explicit synchronization code. There are other opportunities in this space as well, in the form of languages and frameworks for concurrent programming.

Aspects and distributed systems. We also have existing courses on distributed systems. Distributed systems are

yet another area where OOP has been of limited success. While client/server has been well addressed by systems such as CORBA (Common Object Request Broker Architecture) and RMI (Remote Method Invocation), a number of problems have not been solved elegantly within such frameworks. In particular, fault tolerance, fault detection, eliminating central points of failure, dynamic discovery, etc. are examples of *concerns* not easily addressed, again because these issues tend to span multiple (if not all) components in a distributed system.

Industry. Industry applications. Industry is actually playing a vital role in much of the academic discussion. Xerox PARC (whose researchers were among the early proponents of OOP) is where much of the AOP work is taking place. By the time we develop this course, there is likely to be significant interest even in industry for aspect-oriented programming [5].

3.3 Integration

Integration is a key part of our prospective course. AOSD, as we have articulated, clearly has pervasive implications on the way software development is done. To avoid repeating the past mistakes of rolling out OOP without a careful examination of its implications across the discipline, we will not limit the discussion exclusively to the subject of AOP (aspect-oriented programming). The following courses are candidates for introducing AOP early and often:

Introduction to OOP. This course is the first course our students take after CS2. At that point, students are already familiar with the principles of data abstraction (classes and modules). This is the first opportunity to explore OOP at a deeper level. The authors have successfully introduced AOP as a module toward the end of an object-oriented development course for advanced undergraduates and master's students.

Operating Systems. Operating systems is widely agreed as an area where aspects could really make a difference. As this course is usually taken after Introduction to OOP, this would be a place to introduce applications of aspects. The focus of this course is conceptual with some implementation to understand OS programming interfaces, so there is an opportunity here.

Software Engineering. One should not forget that the gap of OOP being transitioned to education was also manifested in industry (with an even longer gap; many industry projects are yet to adopt OOP). By introducing aspects into the software engineering course, students will learn how to capture aspects using UML, a result that can be transitioned from the relationship with one of the author's research group, which has an active project on UML and AOSD.

Extreme Series Courses. Two of the authors both teach a number of topics courses as part of an "extreme" software development series. Aspects are relevant in two of these courses, including concurrency and distributed systems (both of which were addressed in the curriculum topics discussion). Some of the material developed for the prospective undergraduate course is likely to be expanded upon in these courses, which are electives.

4. CONCLUSIONS AND FUTURE WORK

AOSD is not a mere fad and is here to stay. There is a strong community behind it at `http://www.aosd.net`, and a number of significant industry movers are beginning to take it seriously. The SIGCSE community must do whatever it can to ensure that AOSD principles become part of the regimen of software development topics [3] covered as part of an undergraduate education. Furthermore, steps must be taken now to ensure that AOSD does not experience the same lag that accompanied object-oriented programming.

In this paper, we have presented a vision for introducing AOSD in the curriculum and integrating it with existing courses the authors' institutions, and we briefly described our early experience with this integration. We believe this blueprint and preliminary experience will be of value to those who wish to introduce AOP into their curriculum in an evolutionary manner, while at the same time providing an opportunity to be revolutionary by offering a full course.

5. REFERENCES

[1] Mehemt Aksit, Siobhan Clarke, Tzilla Elrad, and Robert Filman. Aspect-oriented software development, 2003. To appear.

[2] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] Alan Fekete and Bob Kummerfeld. Design of a major in software development. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 73–77. ACM Press, 2002.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[5] Gregor Kiczales. Aspect-oriented programming (aop), 2003. TheServerSide.com Techtalk Interview.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

[7] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[8] John Lewis. Myths about object-orientation and its pedagogy. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 245–249. ACM Press, 2000.

[9] Harold Ossher and Peri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

[10] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. Jac: A flexible framework for aop in java. In *Proc. 3rd Intl. Conf. Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, 2001.