A First Course in Software Engineering for Aerospace Engineers

Kristina Lundqvist, Jayakanth Srinivasan
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
{kristina, jksrini}@mit.edu

Abstract

Software is a critical component of mission capability in all aerospace systems. This capability is realized directly through the use of onboard software, and enabled through the use of software on ground support systems. Students attending an aerospace engineering program come with a highly diversified background in software development ranging from novice user to expert programmer. A first course in software development has to account for the diversity, and as an outcome provide both a common vocabulary, as well as a common baseline of skills. This paper presents our learning from designing and teaching such a course for aerospace engineering undergraduates.

1. Introduction

Aerospace systems are ever increasingly becoming dependent on software components. The Task Force on Defense Software [11] estimates the percentage of capability delivered by software in a modern combat aircraft, such as the F22, to be 80%. The existing curriculum in aerospace engineering does not effectively address the need for engineers who are well versed in both the foundational disciplines in aerospace such as fluids, structures and propulsion as well as software design, development and sustainment. The department of Aeronautics and Astronautics at MIT made Aerospace Information Engineering a strategic thrust in 1998 [4] to address this urgent need. Since the decision was made, a number of courses in Aerospace Information Engineering have been offered in the department, ranging from digital communication to artificial intelligence. There was however, a lack of a "first" course that would provide a common launch platform for the students on which they could build more detailed knowledge. There were attempts made to offer a foundational course in the three years prior to the course described in this paper, but they had mixed success.

In spring 2003, the authors took responsibility for the course, and carried out a major revision of both the contents of the course, as well as the teaching methods used. During a series of retrospectives over the previous offerings of the course both with faculty and students, some of the critical issues that had to be addressed were identified:

- The curriculum requirements for freshmen at the MIT School of Engineering do not include a mandatory course in programming. Hence, Aero-Astro students came into the course with a highly diversified background in software development ranging from novice browser user to expert programmer.
- Previous offerings had used a Handy Board for robot programming as part of the course. While almost all the students found building robots exciting, a majority found getting their robots to work as expected quite frustrating, as they did not have a strong background in low-level real-time programming.

Proceedings of the 19th Conference on Software Engineering Education & Training (CSEET'06)
1093-0175/06 $20.00 © 2006 IEEE

- Contextualizing software development in the aerospace context was challenging as 'simple' domain examples do not capture either the complexity of aerospace software or illustrate the power of concepts like tasking.

There are three major contributions of this paper: first, we define the elements of a first course in software engineering for undergraduate aerospace engineers based on the course we taught entitled "Introduction to Computers and Programming;" secondly, we share our learning from using 'pair teaching', code skeleton laboratories as the basis for problem sets, as well as recitations and muddiest point of the lecture cards [8]; finally, we discuss the use of term projects to contextualize the impact of software in the aerospace industry.

There has been significant work done earlier in terms of defining a first course for non-CS majors. Shannon [12] presents a first course based around Lego robot programming using python that can be used for both CS and non-CS majors. Powers [10] presents a two course sequence around Java, however, given the limited use of Java in the aerospace environment, porting the course structure into our context would have been challenging. Guzdial and Forte [7] recently presented a design process for creating a first course for non-CS majors, and we found a strong correlation to the approach we had used.

The remainder of this paper is organized as follows: in section 2, we discuss the overall design of the course, and present the learning objectives as well as assessment methodology used in the course; in section 3, we discuss the actual execution of the course, and highlight critical learning points over the semester; section 4 presents the lessons learnt both during the semester, as well as at the end of the course.

2. Course Design

The course contents were decided based on three factors: an analysis of existing offerings within the School of Engineering at MIT; Discussions with industry partners and senior faculty within the department, to determine the knowledge base that we wanted to build; and finally a study of existing educational standards, specifically the Computing Curricula Computer Science report [2], and the Ironman edition of the SWEBOK [6].

2.1. Course Content

There are six possible approaches to teaching a first course in computing: Imperative-first, Object-first, Functional-first, Algorithms-first, Hardware-first or Breadth-first, depending on the emphasis and sequence of courses taught. Existing introductory courses within the School of Engineering fell into the first four categories. These courses however did not match the needs of a broad breadth-first introduction that was needed for aerospace engineers. Denning et al., [3] recommend in the "Computing as a Discipline" report, that "the first courses in computer science would not only introduce programming, algorithms, and data structures, but introduce material from all the other subdisciplines as well," making sure that "mathematics and other theory would be well integrated into the lectures at appropriate points." This view was strongly recommended in the original Computing Curriculum 1991 report [1], but successful adoption of such a breadth-first course beyond the originators of the course itself have been limited [2]. The elements of the computing curriculum chosen as part of our breath-first course are shown in Table 1.

The SWEBOK defines ten knowledge areas that cover the complete software lifecycle: Requirements, Design, Construction, Testing, Maintenance, Configuration Management, Engineering Management, Process, Methods & Tools, and Quality. From the perspective of our breadth-first course, the teaching focus was on the core lifecycle

phases of Design, Construction and Testing. The enabling components of Requirements, Processes and Management were illustrated through in class case discussions, as well as problem sets, quizzes and the end of term project.

CS Body of Knowledge [2]	Elements Covered in 16.070
Discrete Structures	*DS2* Basic Logic, *DS3* Proof Techniques, *DS5* Graphs and Trees
Programming Fundamentals	*PF1* Fundamental Programming Constructs, *PF2* Algorithms and Problem Solving, *PF3* Fundamental Data Structures, *PF4* Recursion
Algorithms and Complexity	*AL1* Basic Algorithm Analysis, *AL3* Fundamental Computing Algorithms, *AL5* Basic Computability, *AL7* Automata Theory
Architecture and Organization	*AR1* Digital Logic and Digital Systems, *AR2* Machine Level Representation of Data, *AR3* Assembly Level Machine Organization, *AR4* Memory System Organization and Architecture
Operating Systems	*OS1* Overview of Operating Systems
Programming Languages	*PL1* Overview of Programming Languages, *PL3* Introduction to Language Translation, *PL4* Declaration and Types, *PL5* Abstraction Mechanisms
Software Engineering	*SE1* Software Design, *SE3* Software Tools and Environments, *SE4* Software Processes, *SE5* Software Requirements and Specifications, *SE6* Software Validation

Table 1. Mapping 16.070 onto the Computing Curriculum 2001

2.2. Selecting the Programming Language

There is no consensus among computer science instructors about which language is best suited for CS1 and similar introductory programming courses. The choice of programming language should not matter much since many students are at a stage where they are still trying to master the basic fundamentals of programming. The emphasis should be on problem solving, algorithm development, logic thinking, and have good support for software engineering concepts instead of the advanced features of the language. Most CS1 classes use either Java or C++, and while these languages are very effective for implementing large non-critical applications, their use in mission-critical aerospace applications has been limited. Ada 95 has been successfully used in the aerospace domain, and had a strong following in schools with aerospace programs [5]. This teaching mass has since declined as Ada is no longer mandated by the US Department of Defense as the programming language of choice for developing software. However, there have been sustained efforts at the USAFA to continue to use Ada as the introductory programming language [13].

Given that students taking the course will be working on large scale aerospace systems, our requirements for selecting a programming language were based around:

- A history of successful use in large scale, mission critical aerospace applications.
- Compilers that enforce good development practices, strong typing and informative pre-runtime error detection.
- Coding standards that accepted industry practice.

- Language features that are necessary for flight certification (for example by the FAA).

Ada 95 as a programming language met those requirements, and open source GNAT compiler coupled with the AdaGide IDE provided a stable development environment.

2.3. Course Structure

The course was initially designed as a 12 unit course (which imposes a weekly workload of twelve hours a week on the student) in a 3-0-9 format (three hours of lectures, no lab hours, and nine hours of study outside the class room). The absence of laboratory hours was based on an assumption that lectures would contain sufficient code skeletons that students could use for their learning. This assumption required us to rethink how the course was taught, and is further elaborated in sections 3 and 4.

The course was taught using 37 lecture hours, 11 recitations and one end of term project. The lectures were portioned into six segments: Computer organization and architecture, Ada 95 constructs, Discrete Structures and Algorithms, Theory of Computation, Software Engineering, and Introduction to Other Classes as shown in Figure 1.

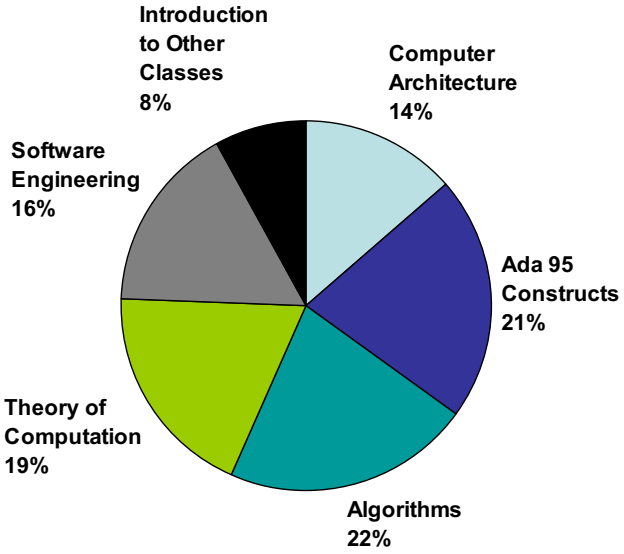

Figure 1. Segmentation in Course Content

2.4. Learning Objectives and Measurable Outcomes

Concisely stated, the learning objective of the course is to provide the students with an understanding of the fundamentals of computer science and to leverage the Ada 95 programming language for problem solving in the aerospace domain. The students were assessed against six measurable outcomes as shown in Table 2. Both formal and informal assessment methodologies were used to ensure performance against those outcomes. The formal assessment occurred in terms of the graded problem sets (both written and programming problems), case studies, quizzes and the final project. More informal assessments were carried out in terms of oral presentations, in class participation and retrospectives during office hours.

Measurable Outcome	Assessment Approach
Solve simple problems in computer science with a specific focus on digital logic, number systems, proof theory, and algorithm analysis	WPS, Q
Gain an intuitive understanding of the process of programming: the cycle of problem understanding, formulation, solution, and implementation	CS, CP, Q, FP
Translate the intuitive understanding to practical implementation using good design practices and software tools	CS, CP, Q, FP, OP
Solve basic programming problems using straight line programs, iterative constructs and recursion	CP, FP, Q
Develop and demonstrate a programming style that is accepted industry practice	CP, FP, Q
Demonstrate an understanding of the impact of computer science on aerospace	OP, ICP, R, Q

WPS: Written Problem Set **CP**: Coding Problem **CS**: Case Study **Q**: Quiz **FP**: Final Project
R: Retrospective **ICP**: In Class Participation **OP**: Oral Presentation

Table 2. Measurable Outcomes and Associated Assessment Approaches

3. Course Execution

The initial design of the course was based around three hours of lecture supported by nine hours of study outside the classroom. As with the best laid plans, there was a lot of tweaking and some outright modifications in the course over the semester. The major elements in the course such as pair teaching, recitations and end of term projects remained relatively static; problem sets and quizzes had to be tweaked in order to meet the pace of learning; and we had to radically alter the class structure to enable the inclusion of laboratory hours. In the subsequent subsections, we will address the three critical elements of the course execution, namely, pair teaching, problem sets & recitations, and laboratories.

3.1. In-Class Pair Teaching

Teaching happens in a turbulent environment that involves a large number of people providing real-time feedback. Like pair programming [9], pair teaching involves a duo of instructors who develop a shared mental model and common awareness of the environment in order to work more effectively. We decided to use pair teaching as the means of leveraging our common knowledge base in computer science as well as to exploit our individual domain expertise in Computer Systems and Avionics respectively. The approach to making the teaching effective was to have one instructor teach the lecture, while the other sat with the students, gathering feedback (observing class reaction, and scribing questions asked/issues raised) as well as leading case discussions (primarily asking leading questions around concepts). There is an initial overhead involved in synchronizing mental models and a continued overhead in terms of pre-lecture meetings, real-time lecture involvement and post-lecture debriefs; however the benefits far outweigh the overhead, as problems get identified and often addressed by the end of every lecture.

3.2. Problem Sets and Recitations

The problem sets were designed prior to the start of the course. We had not accounted for schedule slippage in terms of material we actually covered in class in a week, and minor adjustments had to be made to account for the same. As the problem sets evolved to more complex programming assignments, we saw a significant increase in the time students spent on actual coding of the solutions (we tracked both grades and time spent per problem within a given problem set on a weekly basis). On closer examination of the total time spent on the problem set as a sum of time spent in design and time spent in implementation, and correlating them to scores received for the design section of the problem set, we found two clusters:

- Students that had high scores, and had spent the expected amount of time on the design section, and a significant amount of time on the actual implementation.
- Students that had low scores, and had spent very little time on the design section, and most of their time on the actual implementation.

We did a retrospective in the classroom, and identified that the students who scored poorly in the implementation section were struggling with a combination of programming language syntax, coupled with compilation and debugging tools usage. The creation of optional weekly laboratory hours and mandatory recitations addressed this issue. To account for the students who were creating very limited designs, we addressed the problem in two ways: Firstly, the grading system was modified to provide increased weighting for the analysis and design component in both problem sets as well as quizzes, and secondly problem solving approaches were discussed both in recitations and as part of the weekly laboratory hour.

3.3. Laboratories

There was an implicit assumption made that the code skeletons provided in the lectures would provide the needed foundation on which students could build on for both their problem sets and their end of term projects. We rapidly found that given the varied programming experience, and almost complete lack of any awareness about Ada 95 prior to attending the course (see Figure 2), some mechanism was needed to enable the students to use the code skeletons presented in the lectures as part of the basis for solving the weekly problem sets.

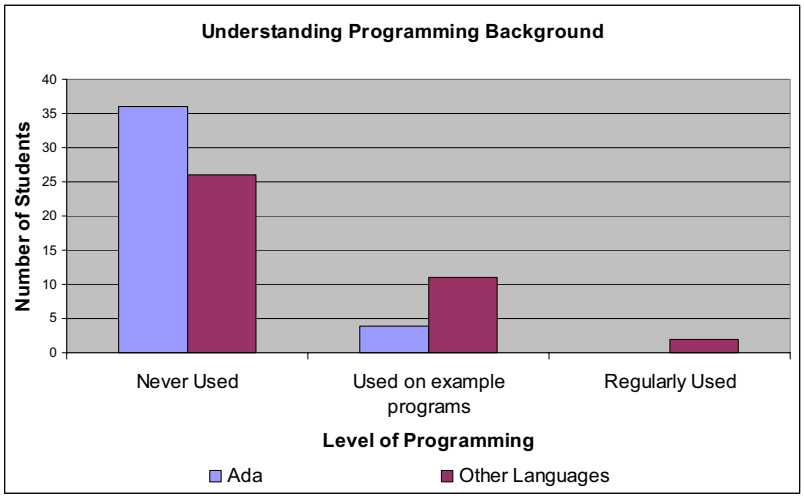

Figure 2. Experience Levels Using Programming Languages

Since the addition of laboratory hours took place after the course had already started, we created four self-organizing teams that would use the lab depending on their schedule openings. Each team was assigned two undergraduate teaching assistants who were responsible for handing out the laboratory questions, the skeleton code created by the course instructors, and at end of the session, provide them with at least one complete solution to problem set. The laboratory problems were timed to take 45 minutes, leaving 15 minutes for questions and discussions with the undergraduate teaching assistants. In addition to enabling the learning of the students, the questions were designed to be used as a basis for solving the problem set.

4. Lessons Learnt

There were multiple feedback loops put into place to capture lessons learnt both while the course was being taught, as well as at the end of the course. An initial survey was carried out to get an understanding of the background knowledge possessed by the students in computer science and programming. This survey allowed us to reduce the impact of some of the problems that we faced during the teaching of the term. A standard end-of-term survey was carried out to assess the effectiveness of the teaching and learning strategies employed throughout the course. Of the 38 students that took the course for credit, 22 completed the survey and provided both quantitative as well as qualitative assessments of various aspects of the course. A summary of the most and least effective teaching and learning strategies is shown in Figure 3.

Two questions that were not included in the standard survey were assessments of pair teaching and retrospectives. Pair-teaching enabled us to leverage complementarities in our backgrounds and teaching styles, however it required both instructors to be on the same page. The pre-lecture meeting ensured that near term objectives were clearly understood, and allowed us to refine the lectures and cases to address specific issues. The post-lecture briefing after every lecture allowed us to identify gaps in knowledge and conceptual misunderstandings, as well as to address them proactively in the next lecture. The retrospectives were used as part of in-class discussions with the entire class, as well as in small teams/individually in office hours. The in-class retrospectives allowed students to share opinions as well as learning with their peers. The learning was often based on the trade publications and academic papers that the students read to successfully scope their final projects.

As seen in Figure 3, the “muddiest part of the lecture” cards were not effective, as the class was highly interactive, and more often than not, a case example or the recitation effectively addressed the questions. The added laboratory hours were deeply appreciated by students who did not have prior programming experience, and while there were some logistical issues with getting the laboratory hours to run smoothly, we believe them to have been instrumental in enabling the learning of our students.

The students were offered a choice of ten projects, all chosen from either currently ongoing or recently completed aerospace systems. One of the authors created scoped versions the systems in terms of project descriptions, and allowed the students to refine the scope by performing a literature review covering academic papers as well as trade journals. These projects were extremely effective in both getting across the impact of software in aerospace, as well as enabling the students to gain a deeper understanding of programming. One student quote in the qualitative segment of the survey said “*the project was good because I actually learned a lot about Ada during it*”.

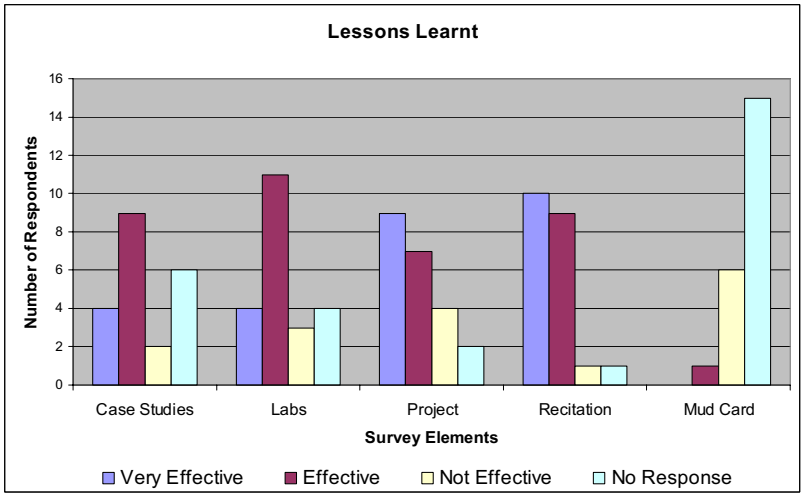

Figure 3. Survey Elements Impacting Lessons Learnt

5. Acknowledgements

The authors would like to thank the students and undergraduate TAs that were part of the first offering of the course in the form of 16.070 – Introduction to Computers and Programming. Additionally, we take this opportunity to acknowledge the three anonymous reviewers for their feedback.

6. References

[1] "Computing Curricula 1991", ACM/IEEE-CS Joint Curriculum Task Force, ACM Press, New York, 1991.

[2] "Computing Curricula 2001 Computer Science", The Joint Task Force of Computing Curricula IEEE Computer Society Association of Computing Machinery, Dec. 2001.

[3] Denning, P.J., P.R. Young, "Computing as a discipline", Communications of the ACM, 32(1):9-23, Jan. 1989.

[4] Department of Aeronautics and Astronautics Strategic Plan, Massachusetts Institute of Technology 1998, available at http://web.mit.edu/aeroastro/www//about/index.html

[5] Feldman M.B., "Ada Experience in the Undergraduate Curriculum", Communications of the *ACM, Vol. 35, No.* 11, Pp- 53-67. 1992.

[6] "Guide to the Software Engineering Body of Knowledge (SWEBOK)", Software Engineering Coordinating Committee, Version. A project of the IEEE Computer Society, 2004.

[7] Guzdial M., A. Forte, "Design process for a non-majors computing course", Proceedings of the 36th SIGCSE technical symposium on Computer science education, February 23-27, 2005, St. Louis, Missouri, USA, 2005

[8] Mosteller, F. "The 'Muddiest Point in the Lecture' as a Feedback Device," On Teaching and Learning: The Journal of the Harvard-Danforth Center, Vol. 3, 1989, pp. 10-21, 1989.

[9] Nosek, J. T. "The Case for Collaborative Programming". Communications of the ACM. March 1998**:** 105-108.

[10] Powers K.D., "Breadth-also: a rationale and implementation", Proceedings of the 34th SIGCSE technical symposium on Computer science education, February 19-23, 2003, Reno, Nevada, USA, 2003.

[11] "Report of the Defense Science Board Task Force on Defense Software", Office of the Under Secretary of Defense for Acquisition and Technology, Washington, D.C. 20301-3140, Nov. 2000.

[12] Shannon C., "Another breadth-first approach to CS I using python", Proceedings of the 34th SIGCSE technical symposium on Computer science education, February 19-23, 2003, Reno, Nevada, USA, 2003.

[13] Sward R.E., C. Martin, B. S. Fagin , D. S. Gibson, "The case for Ada at the USAF academy", Proceedings of the 2003 annual international conference on Ada: the engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies, p.68-70, December 07-11, 2003, San Diego, CA, USA, 2003.