Proceedings of the 36th Annual Meeting of the Decision Sciences Institute
DSI-2005, San Francisco, California, pp 1851–1856.

Introduction to Programming and Software Development: an Interactive, Incremental and Iterative Approach

Raja Sooriamurthi, Information Systems Department
Kelley School of Business, Indiana University, Bloomington, IN 47405
(812) 855-4254, raja@indiana.edu

ABSTRACT

How does one teach programming to a student who does not necessarily intend to become a hard-core programmer? With this question in mind, over the past two years our introductory course on programming and software development for business majors has been completely revamped and restructured. This paper discusses our experience and the effectiveness of our innovations in attaining our pedagogical goals. This work has been supported by three curriculum development grants from Indiana University over the past two years.

Keywords: Application Development, Programming, Pedagogy, Active Learning

INTRODUCTION

The training and career goals of a student of information systems are similar to but different from that of a student of computer science. As a discipline, information systems focuses on the blend of business, technology and people. A primary perspective of the field is to examine how technology can enhance and add value to a business. Hence a student of information systems needs to be trained in how technology can be used as tool. In that respect, software is one of the most powerful tools modern practitioners of information systems have at their disposal. The goals of our introductory course on programming and software development are to introduce students to work with software, to study existing software systems, and to lay the foundation to build and modify software of their own.

Course Objectives

To lay the background and motivation for the pedagogical enhancements and methodology described in this paper let us first consider the objectives of our introductory course on programming and software development:

1. First and foremost, to cultivate the process of algorithmic thinking
2. To learn to express algorithms in a modern language such as Visual Basic.net, C# or Java.
3. To gain an introduction to the process of software development
4. To understand the foundations of the object-oriented paradigm
5. To gain an appreciation of the principles of good design that transcends all design activity: abstraction and modularity

6. To gain a better appreciation of the bigger picture of technology in the real world.

The above objectives are to be attained in the overall framework of how an information system can enhance the productivity of and add value to a business.

PEDAGOGICAL INNOVATIONS

With the above goals in mind the implemented pedagogical innovations have been summarized below:

- ***A blend of algorithmic reasoning and the concepts of object-orientation.***

  Object-oriented programming and design has evolved to be the current major software development paradigm [1, 2]. But first any software developer needs to master *algorithmic thinking*. In this course students are introduced to the concepts, terminology and the object-oriented frame of mind from day one. At the same time they get an effective blend of algorithmic reasoning as more and more concepts of object orientation are gradually introduced.

- ***Restructuring of the course into a lecture + lab format.***

  The earlier version of this course (S205) consisted of two 75 minute lectures. To emphasize learning by doing, I restructured the new course (S308) to have two 50 minute lectures and a 75 minute lab. To facilitate greater instructor student interaction, students in a single lecture section (25–30 enrollment) are distributed across two lab sessions (each with 12–15). I have also developed an extensive set of lab exercises described later in this paper.

- ***Emphasis on the reading and writing of programs.***

  One becomes a good writer by writing *and* reading exemplary works of literature. Similarly, one becomes a good software developer by writing and reading exemplary programs. A significant section of the software industry deals with code maintenance which requires the ability to read and comprehend programs written by others. To develop these skills, all the assignments and lab exercises model a typical industry development scenario by consisting of two components: (1) read and understand a given code outline (2) complete and extend its functionality by writing novel code.

  - ***Model solutions:*** Over the course of the semester I have personally written 6000+ lines of code in the form of solutions to assignments, lab exercises and class examples. To ensure that I am able to effectively address all student questions, I personally develop solutions to all the homework assignments and lab exercises before issuing them. I am thus intimately familiar with the design of the software and am aware of the most common mistakes a student might make.

- ***An interactive, iterative, incremental approach.***

  Very early in the course students are introduced to the power of the Visual Studio.net debugger. They are encouraged to make full use of the debugger in exploring their evolving solutions. Students are also guided towards getting a basic working version (version 0) of their program before extending it incrementally further. In one of their advanced assignment they are required to submit incremental versions of their working code.

- ***Innovative use of analogies, metaphors and graphics.***

  Learning takes place at the fringe of our existing knowledge. I believe the key to being an effective teacher is to establish a link between what the students already know and what they need to know. I have found that the foundational concepts of programming and object-oriented design can be made more intuitive when explained with appropriate analogies and metaphors. I have created numerous analogies and examples that link the new concepts I am teaching my students to what they already know. Illustrations may be found in my earlier papers at the Innovation and Technology in Computer Science Education (ITiCSE) conferences [3, 4].

- ***Emphasis on the bigger picture of information systems in the business world.***

  Information systems is an exciting and fun field! To provide an opportunity for students to explore a topic of their choice as well as to develop their presentation skills, all students are required to make a 20 minute oral presentation. Students' presentations cover a broad spectrum of topics in information technology such as genetic algorithms, online music, encryption, Internet-2, satellite radio, virtual reality etc. Presentations are video taped (with the assistance of campus media services) and part of the grade was based on peer evaluation of other students in the class. As an alternative, I have also required students to write an essay profiling an influential person in the IT field. Based on a writing/teaching grant I have received I have also incorporated a written component into the course wherein students write a 4–5 page executive summary of an emerging technology.

COURSE STRUCTURE, CONTENT AND STYLE

The focus of the course is on the *practice* of software construction. The course philosophy may best be understood by the following comparison: Just as one may not necessarily be conversant with the nitty gritty details found in a drivers manual but can still be a very good driver, I want students graduating from my class to be skilled in the practice of developing software and not necessarily the nitty gritty details of a programming language. Any language manual, combined with a good development environment will provide language specification support. But my students need to know the the fundamental principles of good design in general and good design in software in particular. The content, pedagogical style and performance evaluation of the course emphasizes this view of how to write modular software using proper abstractions.

Organization and Content

The course is organized around two 50 minute lectures and a 75 minute lab. The course content is primarily assignment/problem driven — the knowledge and skills needs to complete each assignment typically forms the basis for the lecture discussions. Over the course of the semester students work on 5–7 programming assignments requiring around 200 to 600 lines of code. I provide a code/design outline (typically about half the total amount of code) and the students need to write the remaining. Completely working out each programming assignment before hand has the dual advantage of (1) as I am intimately familiar with the requirements of the assignments when a student has a problem I'm able to provide effective guidance and (2) on the day the students submit

their work, I spend the lecture session showing them my complete solution, highlighting some of the nuances and moderating a discussion on some of the more challenging parts of the assignments. Students have found it very beneficial to be able to study my complete working application and to contrast it with their own work. Snap shots of a subset of the assignments students have worked on are given in Figure 1.

The Lab Component

A lecture class of 30 students is partitioned into two lab sections of 15 students each in order to increase instructor-student interaction. Every lab typically has two conceptual components: an exploratory component wherein students primarily focus on learning new concepts and an examinational component wherein students reinforce concepts they are expected to be familiar with. The labs are highly interactive and foster an active learning environment. After discussing the objective of the session, I show the students a complete working application and then give them part of the code. They are required to write the remainder. After an allotted period of time, I incrementally show them my solution which allows them to correct any errors in theirs and then we move on to the next stage. This interactive and iterative way of completing the lab sessions encourages students to think on their own but at the same time does not penalize those who may not be able to complete a sub-task in the alloted time.

EVALUATION OF STUDENT WORK

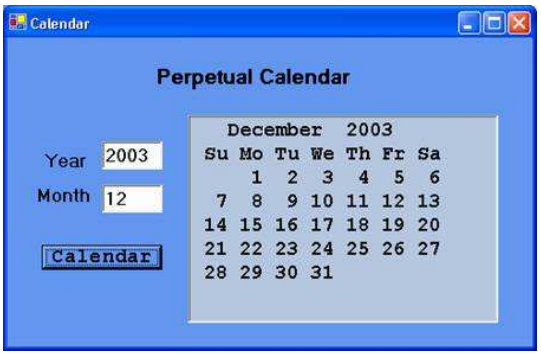
Student performance in the course was evaluated along the following dimensions.

Written exams: evaluation of algorithmic thinking

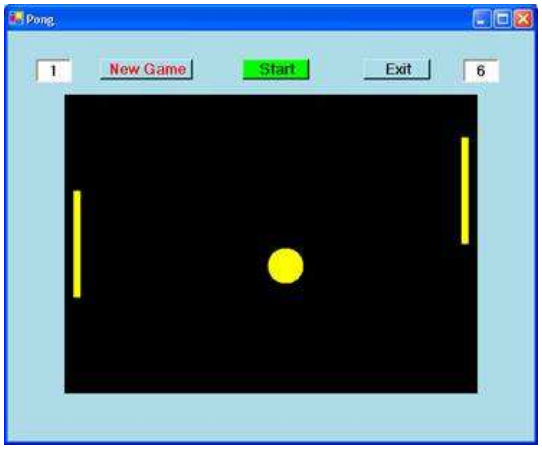
The exams are scheduled for 1.5 hours (midterms) and 2 hours (final). But I give my students *unlimited* time. I tell my students that I do not want any time constraints to influence their performance. My exams are in class, closed book and emphasize algorithmic thinking and hence syntax doesn't matter. But logic does! I set the exam for the allotted time (1.5 or 2 hours) but students are welcome to take as much time as they want. Some students finish in early and some take more than an hour of extra time. Students have favored this approach as it is their reasoning rather than their speed which is being evaluated.

Programming assignments: reading and writing code

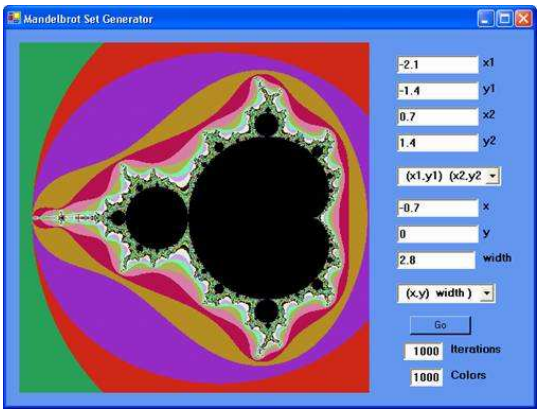
As mentioned earlier, I work out the complete solution for all my assignments so that I am intimately familiar with all that is required. I then take my code, strip out vital portions and provide the remaining outline as a starting point for the assignment. Students need to read and understand my code outline and extend it with their own code. During class I also demonstrate my working solution so that students are aware as to how their own programs should behave. With this model of assignments I try to model a typical industrial development environment where they will need to read and understand existing code before enhancing it further. Over the life span of a software system typically 80% of the cost is for code maintenance — fixing bugs and adding new features. This model of programming assignments encourages the skill of reading programs. It also allows my students to work on significantly larger assignments. By the end of this introductory course they are required to read and extend a program that runs to 600+ lines of code.



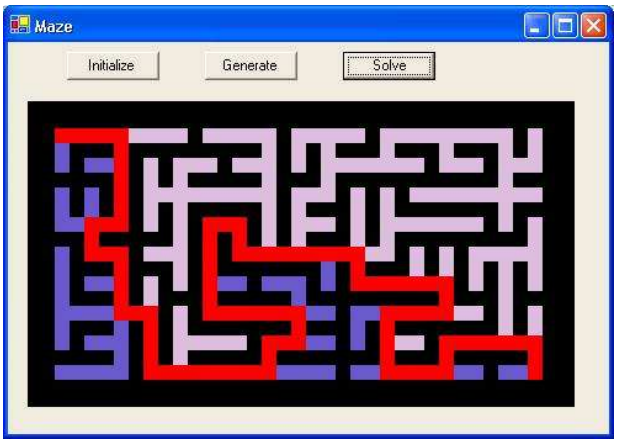
A Perpetual Calendar



The game of Pong



Mandelbrot set generation



Generating and Solving a maze

IFS fractals

A mosaic of colors (an exercise in nested loops)

Figure 1: Screen shots of some sample homework assignments and lab exercises. Graphics form an effective framework for illustrating some of the foundational ideas of programming.

Also, for each programming assignment I provide my students before hand the exact grading slip my grader will be using during the process of evaluating their work. I emphasize the importance of self-evaluation through out the course and ask my students to evaluate their own performance by assigning what they felt they should get for each component of the assignment. They are then able to compare their self-evaluation with what they actually received from the course instructor.

EFFECTIVENESS & TRANSFERABILITY

These ideas have been in development for the past two years (since summer 2003) and I am continuing to refine them. Based on my presentations at earlier conferences other faculty members have borrowed some of these ideas, assignments and lab material for their courses. For the past two years Indiana University has conducted JETT workshops for high school teachers [5]. The Java Engagement for Teacher Training (JETT) workshops (jett.acm.org) were conceived by the ACM and the College Board to help high school teachers transit from a C++ based curriculum to a Java based curriculum in AP computer science exams. The participant high school teachers have taken some of these ideas, examples and code with them to use in their Java classes at their respective schools.

Based on my high course evaluations, increased enrollments in my class, feedback from the high school teachers who have attended my lectures and feedback from students who subsequently did internships or went on to graduate school, this approach to teaching software development seems to be quite effective. For anyone interested, all of my course material is publically available at `http://www.kelley.iu.edu/raja/classes/s308`.

Acknowledgments: Course development grants

I am grateful for the support I have received from my department Chair and school Dean as we gradually introduced these enhancements over the past two years. This work has been supported by three teaching grants which are gratefully acknowledged: Course Development Grant (Kelley School of Business), Active Learning Grant (Indiana University), Writing/Teaching Grant (Indiana University).

References

[1] Robert W Floyd. The paradigms of programming. *CACM*, 22(8):455–460, August 1979.

[2] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[3] Raja Sooriamurthi. Prelude to the java event model. In *Proceedings of the sixth annual conference on Innovation and Technology in Computer Science Education*, page 193, 2001.

[4] Raja Sooriamurthi. Problems in comprehending recursion and suggested solutions. In *Proceedings of the sixth annual conference on Innovation and Technology in Computer Science Education*, pages 25–28, 2001.

[5] Raja Sooriamurthi, Arijit Sengupta, Suzanne Menzel, Katie Moor, Sid Stamm, and Katy Börner. Java engagement for teacher training: An experience report. In *Proceedings of the Frontiers in Education Conference*, 2004.