Introductory Programming with ALICE as a Gateway to the Computing Profession

W. Brett McKenzie
wmckenzie@rwu.edu
Gabelli School of Business, Roger Williams University
Bristol, RI 02809, USA

Abstract

To help reinvigorate the Computer Information Systems (CIS) major in the face of declining student interest, a new curriculum with more welcoming entering courses was implemented at an undergraduate school of business. One of the courses, the introductory programming course, was redesigned to focus on object-oriented, event driven, programming using ALICE, a 3D programming environment developed at Carnegie Mellon University and funded by the National Science Foundation (NSF). This presentation addresses the rationale for selecting ALICE, the subsequent course design, and expectations for the future course developments.

Keywords: computer programming curriculum, Alice, introductory course description

1. INTRODUCTION

To combat the decline in enrollments in computing fields, the faculty redesigned the Computer Information Systems (CIS) curriculum for the academic year 2005 to provide a set of core courses and two tracks, one in web and systems development and the other in networking and security (McKenzie, 2005). The core courses begin in the freshman year with a two sequence introduction. This includes a course in building a computer, which also allows for installing different operating systems (Windows and Linux) as well as configuring a IP network, and an introductory programming course, focused on object-oriented, event driven, programming. These courses have no prerequisites. They are required of declared majors and are designed as an open invitation to the major or minor. They may also be taken as an elective for students with a general interested in computing.

The CIS program is housed in the school of business. Similar to counterparts at other institutions, it has an application focus and usually draws examples and problems from the business domain.

In recent years, however, content areas that were once the purview of CIS alone have been adapted and absorbed by other majors. For example, the HTML and web design courses originated in the CIS department but have since been appropriated by communications and design. Similarly, specialized database courses, such as offerings in Accounting Information Systems or data mining for Marketing courses, have cannibalized the CIS database courses. These more pragmatic, discipline centered courses, have made CIS offerings seem more arcane, technical, and distant from day to day realities. In this environment, a secondary goal of the introductory sequence was to assert the relevance of computing from the perspective of inventive producers rather than mere consumers of computing. This paper focuses on the programming course, providing a rationale for the selection of the programming environment, a description of the course, and observations on the student projects.

2. BACKGROUND

Teaching programming is central to the computing curricula. Whether it is the more theoretical computer science (CS), more hardware focused computer engineering (CE), or the more application centered computer information systems (CIS), the governing curricula documents require instruction in computer programming (Shackleford, R. et. al. 2005). While the number of recommended programming courses differs among the various curricula from minimums of two in the "Information" fields to four in the "Computer" fields, the sequencing of computer programming in the first or second years usually makes programming the entry point to the profession.

Perhaps even more important than the curricula requirements are the perceptions of prospective students, who see computer programming as central to the field (Carter, 2006). However, the reputation of programming as socially isolating and hard tends to discourage students. In the IS field, in particular, students express a desire to work with computing, but in areas other than programming. Anecdotally, among the recent offerings at the college, the networking security and forensics courses are oversubscribed while the traditional courses in programming, database, and systems development struggle to meet minimum enrollments.

The course designers needed to consider the incongruous forces which recognize that programming is central to the computing discipline, yet, in practice, many students avoided the programming courses. In this environment, the course designer must ask whether programming courses should be the gate-keeper course to weed out the undesirables, similar to organic chemistry for the pre-med student, or should the programming course be structured as a compelling gateway to a still developing and critical profession? To answer the question, the course designers examined the traditional approach to teaching programming and proposed an alternative approach that might be more relevant to the skills and experiences of twenty-first century students who have grown up with computing, gaming, and the interactive, graphical user interface.

3. TRADITIONAL APPROACHES

Computing courses typically begin with technical details and on a small scale. In many ways, they follow a behaviorist paradigm, where the solution steps are more important than the content of the problem. At the same time, these small problems diminish the central feature of massive scale that makes computers so critical to complex tasks such as DNA sequencing or data mining. The features of a traditional approach to teaching programming are below and can be confirmed by skimming the table of contents of common entry level texts in .NET or C++.

First, many of the early programs are trivial and painfully more challenging for a novice to write than simply figuring out the results. Common entry level programs in schools of business involve money, such as creating a cash register that will multiply quantity of purchase by the cost of items, add the purchases to get the total, compute the tax, then display a grand total. Programming the solution is much more complex than "working it out in your head" as many students are capable of, as witnessed by their being able to compute a bar bill close to closing time.

Second, many of the entry level programs are decontextualized. In the example above, the usefulness of the cash register program is within the larger context of a store where it can track multiple purchases that become too burdensome to track manually. The register is used both to assist in more quickly computing the total sale as well as keeping the cashiers honest (the original motivation for the bar owner, James Ritty, to invent the cash register). Cash registers also now assist in keeping the business honest as the register provides a record of the sales for the tax authorities. In a more complex instantiation, the register serves as a conduit to a database, to query and post a price derived from a UPC code and to update the status of inventory. Forcing students to program a single part of a system, without explanation of its purpose and merely as a demonstration of the task, frequently serves to frustrate students and make the problems seem trivial.

Third, model programs often draw from mathematical areas, and usually include probability problems. These problems often

require the student programmers to examine their own mental process for tasks they may have ill learned or not learned systematically. Computing percentages, for example, can challenge a student's grounding in either ratio arithmetic or decimal representation. It is surprising how few students can explain why a 10% discount is the same as 90% of the original, although many are capable of arriving at the solution with the rationalization, "It just works like that."

4. ALTERNATIVE APPROACH

The mathematical-logic problems are the most common in the traditional programming language instruction, although computer capability has long outstripped its nascent tabulating function. Consequently, this approach misses the critical perspective that computer programming is a means of representing the world (Graham, 2004). I use the term *re-present* rather than represent because *re-present* emphasizes that computing may be thought of as a medium to provide a unique representation of the world. Just as movies derived from novels based upon the real world represent different views of the same events, it is helpful to think of computing as providing another perspective on events.

For example, the cash register program alluded to earlier, *re-presents* the day's transactions in a store, just as the simulation of a craps game *re-presents* the throw of dice in its output. Unlike the rich natural language and the interpretive power of the brain, the *re-presentation* of a computer program uses a language of limited expressive power which is executed in an explicit environment. An adept programmer can create a richer world than a novice can imagine, just as a visual artist can render a scene with a palette of a few selected colors. But programming is rarely introduced as a means of *representing* the real world.

In lieu of using a traditional language, such as JAVA, dominant in CS curricula, or Visual Basic, long a standard to introduce computer programming in business schools (Haney and Lovely 2003), the revised curriculum uses ALICE (Alice, 2003). ALICE is a 3D programming environment where the objects are characters (human, animal, or fantasy) that inhabit worlds (real or imaginary). Programming in ALICE can be either as an animation, for example telling a story through the character's actions, or interactive, as in a computer game. The notion above of re-presentation is embedded in ALICE as the ALICE worlds recreate a real world or a world of its own.

In addition to ALICE providing a compelling environment for students who have grown up in the "gaming generation" (Beck and Wade, 2005), writing code is via dragging and dropping tiles into an interactive editor (Fig 1). The tiles often contain modifiers and areas to include arguments. This provides an easier access to the programming environment than typing and echoes the intelligent editors, such as those in Microsoft Visual Studio which use Intellisense.

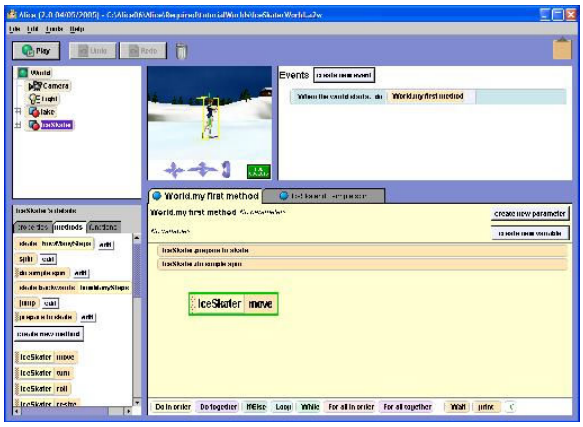

Figure 1: ALICE Development Environment showing (from top left to bottom right) Toolbar, Object Tree, World Window, Events Editor, Detail Area, Program Editor

The challenges of learning the syntax account for some of the difficulty in programming. The drag and drop method has been reported as an advantage in using ALICE (Moskal et. al. 2004). However, as Radneski (2006) demonstrates, even a simple text program in JAVA requires not just an understanding of the syntax, but to stretch the metaphor, also the grammar. A simple JAVA program beings with either students glossing over the meaning of the first line "public class ClassName" to get to writing the first method or it requires extensive explanations prior to any programming as shown in Figure 2.

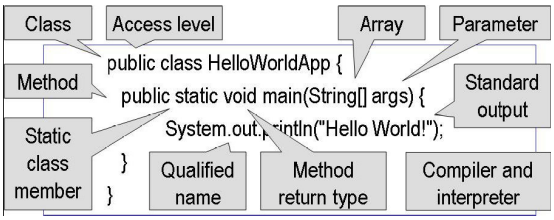

Figure 2: A minimal JAVA program (from Radenski, p. 197)

In comparison, Figure 3 shows the code and the result for a Hello World program written in ALICE where a penguin character provides the output.

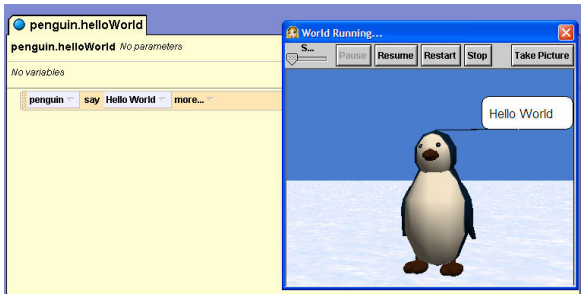

Figure 3 Hello World in ALICE

The Penguin class comes from the collection of characters. A new method is invoked and named penguin.helloWorld and a single line of code, created with a tile for a primitive method, "say", which contains in itself the argument, completes the program.

The ease of running a program in ALICE complements the ease of programming, due to the reduction in complexity of the syntax and more explicit nature of the grammar, and facilitates debugging. For example, when a character's wing detaches from its body because a "move" method rather than a "turn" method is called, students immediately recognize the error. Unexpected behaviors in ALICE worlds, such as the challenge of getting a ball to roll realistically by considering both rotational and translational motion, encourage students to consider the difficulties of representing the real world. Techniques, such as problem decomposition and step-wise refinement, fall naturally from resolving these issues.

5. COURSE DESIGN

Following current trends, the course is used to introduce object-oriented programming concepts and event driven programming. The course was designed in a modular fashion, where the first module focused on program planning, introduced the ALICE programming environment, and basic control structures such as conditional statements, functions and expressions. The second module centered on classes, objects, inheritance, and methods as well as events and event handling. The final two modules examined loops, recursion, arrays, lists, and advanced debugging techniques. A final section was included to introduce students to Visual Studio so that they would have some familiarity with the future programming environment and be able to apply the concepts from ALICE in a traditional language.

The major assignments included three student projects. Following the first module students completed a solo project to create an animated card (similar to a Blue Mountain e-card). To complete the second module, students worked in teams to create an interactive experience, generally a game. Finally, students completed a third project, either as a team or alone, that could be either an animation or an interactive experience. Each of the assignments had broad expectations, such as at least one instance of a new class, one user defined function, and one control structure, but did not require all students to complete the identical problem as is common in traditional programming courses. Additionally, each project included a planning process. An evaluation of the planning documents submitted with the program was included in the grade. Students were free to implement the requirements in any context, which they certainly did, producing among other things a St. Patrick's Day card, a beer-pong game, a helicopter rescue game, and an interpretation of "West Side Story" set on the university campus.

While not mentioned in other research and reports on the implementation of ALICE, its ability to import sound increased the appeal for students. Their projects could include either music or their own recorded information. Including audio added complexity, requiring students to consider the timing and pacing within their programs to coordinate with the sound track. Interestingly, there were no requirements to use audio, but students chose to of their own accord. The combination of audio, ease of programming, and freedom to build self-designed programs to meet broad goals resulted in highly engaged students.

6. CRITIQUE

There was some criticism for adopting ALICE, primarily because it is not a "useful" language; rather it is a teaching language. Interestingly, a common justification for selecting a language, when confronted with the criticism that one should teach, for example C and not VB, has long been that the language being challenged is used to teach the larger issues of programming and not to train students in a specific language. Using ALICE highlights the importance of teaching essential programming concepts devoid of arguments for a specific language. The criticism was further blunted by the involvement of the students, the relative complexity of the programs, and the length of the code generated. In writing studies from the humanities, it has been shown the length of writing often correlated with better performance. By similar measure, a working program of longer length and greater complexity exhibits greater engagement by the students.

The second criticism was that ALICE moves students away from business programming. For example, it does not allow for reading from and writing to files, a central aspect of traditional business programming. Close to mid-term, a student expressed concern about its usefulness. Interestingly, they had just completed a programming exercise to cause a randomly swimming fish to swim down if it approached within a certain distance of the ocean surface and up if it were too close to the ocean floor, or to continue moving randomly forward, up, and down. When asked to produce a logic diagram for a program to assist in stock sales or purchase based upon Bollinger bands, all students were able to chart the logic of the nested control structures. This provides some evidence that the students are able to transfer the skills from the ALICE environment to more mainstream problems.

6. FUTURE DEVELOPMENTS

While ALICE has been adopted in whole or in part for a number of CS programs, its use in schools of business seems rare, especially for a whole course. Courte et. al. (2006) reported using ALICE for a programming module in a survey course in CS, but an informal survey of area schools of business does not show any extensive use. The revised programming course sequence now has a full semester of programming in ALICE followed by .NET programming using C#. As yet, it is too early to evaluate the success of ALICE as preparation for the more mainstream C# because the first students to take ALICE are currently enrolled in the follow-on .NET course. A major question will be whether the expected advances in student's understanding of the fundamental concepts will allow for a more accelerated pace in the subsequent course so that the instructor can focus more explicitly on the business domain.

It is expect that the second time the course is offered in Spring 2006, it will be modified to provide a more focused introduction to Visual Studio and a traditional language. In this modification, the modular structure will be retained, and the target language will be introduced at the end of each module to demonstrate its similarity to ALICE. Among the ALICE community, this course design is referred to as a blended course and there is currently work underway on an ALICE/JAVA curriculum model. The proposed ALICE/Visual Studio blended course may find an audience among undergraduate schools of business.

In summary, the revised course with ALICE appeared to be a success based upon the engagement of students and the quality of their programs. Anecdotal comments, such as one student, who noted that his experience with ALICE helped him finally understand both how to diagram and code more complex program branches, have persuaded the faculty that the course appears to be achieving its goals. The most compelling evidence, however, is that all freshman and sophomore students taking the course in spring have elected to take the follow-on programming course this fall.

8. REFERENCES

Alice V2.0 (2005), http://www.alice.org/

Beck, John C. and Mitchell Wade (2004) Got Game, How the Gamer Generation is Reshaping Business Forever. Cambridge, MA: Harvard Business Press.

Carter, Lori (2006) "Why Students with an Apparent Aptitude for Computer Science Don't Choose to Major in Computer Science", SIGCSE '06, 38:1, p27-31

http://doi.acm.org/10.1145/1121341.1121352

Courte, Jill. Elizabeth Howard, and Cathy Bishop-Clark (2006), "Using Alice in a Computer Science Survey Course", Information Systems Education Journal, 4 (87). http://isedj.org/4/87/

Graham, Patrick (2004) Hackers and Painters: Big Ideas from the Computer Age, Sebastopol, CA: O'Reilly Media, Inc.

Haney John and John Lovely (2003), ".NET as a Teaching Tool", Information Systems Education Journal, 1 (16) http://isedj.org/1/16

McKenzie, W. Brett (2005) "Information Systems Curriculum Revision in a Hostile Environment: Declining Interest, Threats from Offshore, and Proprietary Certification" Information Systems Education Journal, 4 (105) http://isedj.org/4/105

Moskal, Barbara, Deborah Lurie, and Steve Cooper (2004). "Evaluating the effectiveness of a new instructional approach", SIGSE '04. http://doi.acm.org/10.1145/971300.971328

Radenski, Atanas (2006), "Python first": a lab-based digital introduction to computer science", ITICSE '06. p. 197-201. http://doi.acm.org/10.1145/1140124.1140177

Shackelford, Russell et. al. (2006), "Computing Curricula 2005: The Overview Report", SIGCSE '06. http://doi.acm.org/10.1145/1121341.1121482