Full paper in Proceedings of the 16th Annual NACCQ, Palmerston North, New Zealand, July, 2003 (eds) Mann, S. and Williamson, A. www.naccq.ac.nz

The Trouble with Teaching Programming

Patricia Haden
Dr SAmuel mann

School of Information Technology and Electrotechnology
Otago Polytechnic
Dunedin, NZ
Phaden@tekotago.ac.nz

ABSTRACT

Introductory computer programming is notoriously difficult to teach, with both research and anecdotal evidence indicating that many students struggle with these courses, and gain only very limited programming skills from them. In the modern curriculum, this problem is exacerbated by the need to acquaint students with a wide variety of programming languages, paradigms and software architectures. At Otago Polytechnic we are currently developing a sequence of introductory programming courses that we hope will provide students with good core programming skills and competence in both the Procedural and Object-Oriented paradigms. The courses are designed to be tractable for novice programmers, while still satisfying the modern end-user's high expectation of what a computer program should do. In this paper we discuss our underlying educational philosophy, give examples of the instructional material, and look briefly at student performance during the first presentation of the course.

1. INTRODUCTION

It is not easy to teach someone to program. Since computer programming first entered the common curriculum, educators have agonised over the poor performance of students in introductory programming courses (Decker & Hirshfield, 1993; McCracken, 2001). The situation has been exacerbated in recent years by the exploding popularity of Object-Oriented Programming. It was difficult enough to teach procedural programming alone; it seems more than twice as difficult to produce students who can use either paradigm. If we insist that our students first acquire traditional programming skills, they struggle with the "paradigm shift" when we attempt to introduce Object-Oriented logic (Guzdial, 1995; Bergin, 2000). If we try to avoid the paradigm shift by starting with Object-Oriented programming, we produce Object-Oriented designers who don't have the basic programming ability required to actually implement the elegant class structures they design (Duke, *et al.*, 2000).

At Otago Polytechnic we are attempting to address these issues with a new two-semester introductory programming sequence that we hope will act as a bridge between traditional procedural programming and advanced Object-Oriented system development. The design of the course rests on the following philosophical principles:

1. Basic programming skills (variable declaration and usage, conditionals, loops and basic data structures) are essential for success with any programming paradigm.

2. The more experience one has with the procedural paradigm, the more difficult it is to shift to the Object-Oriented paradigm.

3. Students are most likely to persevere with a difficult task if the process is fully engaging, and the end result is satisfying.

4. Students will learn most comfortably if new skills build seamlessly upon existing skills.

In this paper we will discuss how we have applied each of these principles to the design of our new programming course. We will detail the use of Microsoft ™Agents, a programming component that we have found especially useful for building programming tasks.

Finally, we will discuss our observations of student performance during the first offering of this course.

2. THE COURSE PHILOSOPHY

Basic programming skills are essential for success with any programming paradigm.

A growing body of evidence indicates that, especially for the design of large systems, Object-Oriented programming is an efficient technique which produces robust and reusable code. However, designing an Object-Oriented system requires more than the production of an elegant UML diagram. At some point the code for the methods must be written, and this involves manipulation of data structures and proper flow of control, which are the core components of the traditional programming course. Recent evidence (Duke. et. al, 2000) indicates that students whose first introduction to programming is in an Object-Oriented environment may fail to acquire these basic programming skills. Duke et. al. state that "in later years, students who have not adequately mastered these basic programming skills (using while loops and Boolean expressions to capture a system's internal logic) may be able to create higher-level designs, but struggle to convert those designs into actual code." (p.84).

To insure that our students have the core skills required to allow them to implement the Object-Oriented systems they will one day design, we start them off with a 12-week introductory course in Pascal (PR104). This course, which closely follows the classic introductory programming curriculum, is taught using Turbo Pascal 7.0 with console input and output. The emphasis of the course is on issues of syntax, handling of variables and simple data structures, and the use of booleans, conditionals and loop constructs to implement correct program logic. Students are not permitted to proceed to the next course in the programming sequence until they have mastered this basic material.

The more experience one has with the procedural paradigm, the more difficult it is to shift to the Object-Oriented paradigm.

Various authors have explored the great difficulty encountered when trying to teach Object-Oriented programming techniques to experienced procedural programmers (e.g. Turk, 1997; Nelson, Armstrong & Ghods, 2002). The generally accepted explanation for this difficulty is that well-learned procedural patterns interfere with the development of new, sometimes orthogonal Object-Oriented patterns via the established psychological phenomenon of retroactive inhibition (colloquially, 'you can't teach an old dog new tricks'). To minimise the impact of retroactive inhibition, we want to expose our students to the concepts of Object-Oriented programming as soon as they are comfortable with the basic mechanics of programming, but before they have become set in their procedural ways. To this end, as soon as students have completed twelve weeks of introductory Pascal training in PR104, they proceed to the second course in the sequence (OO104) where, by easy stages as described below, they are introduced to the Object-Oriented paradigm. With only twelve weeks of training in traditional procedural programming, we believe that students will view Object-Oriented logic as just another new skill set, no more alarming than the syntactic rules they learned in Introductory Pascal.

Students are most likely to persevere with a difficult task if the process is fully engaging, and the end result is satisfying.

There is considerable evidence that students learn most effectively when their learning tasks are engaging – that is, both entertaining and personally relevant (Webster & Ho, 1997; Kearsley & Shneiderman, 1999). This observation applies as much to computer programming as to any other discipline (Guzdial & Soloway, 2000). As educators we must acknowledge that today's students have a different relationship with computers than we had when we were learning to program some decades ago. Accustomed as they are to using modern GUI applications for work and recreation, they have very high expectations for what they should be learning to make computers do (cf. Stein, 1998). Writing "Hello World", or producing a list of the powers of 2 no longer produces the frisson of excitement it might have for those of us who remember programming with punch cards. We have an obligation to enable students to produce programs which give them that same thrill, and we must do so in the context of their a priori experience.

To this end, we teach our second programming course using Borland Delphi. Delphi is an RAD tool comprised of an authoring environment that makes construction of GUI nearly trivial, a robust hybrid Object-Oriented programming language in Object Pascal, and an extensive set of libraries and extensions that insure it will satisfy a student's programming needs for many years.

As Object Pascal is essentially a superset of the Turbo Pascal dialect with which they are already familiar, our students can use Delphi from the first day of the course. With Delphi's simple screen design facility and powerful built-in image manipulation tools, we are able to very quickly have our students completing laboratory tasks that they consider to be "real programs". Figure 1 shows a screenshot of a "Slot Machine" game that students build during the

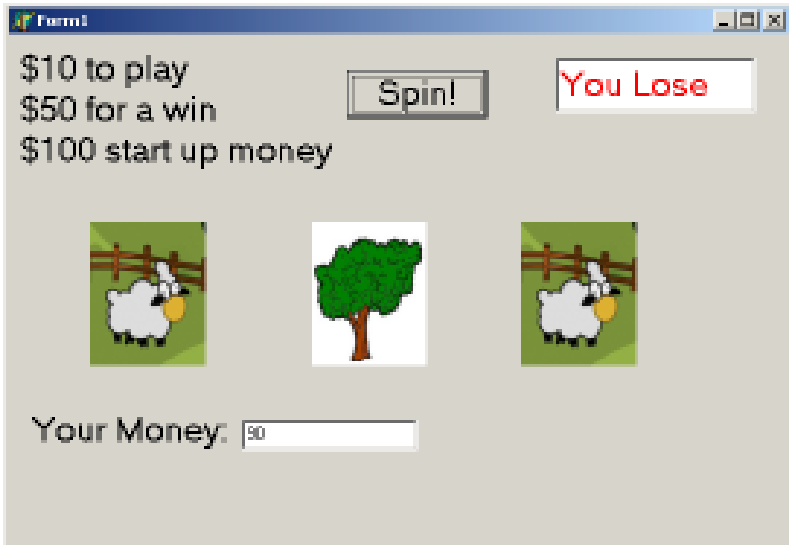

Figure 1. Screenshot Of Slot Machine Game.

second two-hour laboratory session of the Delphi course. Students are able to make the program "spin" the images, randomly assign final images to each location, and keep track of the money won and lost by the user. In our first instantiation of the Delphi course this task was completed successfully by 100% of the students.

All the material in the Delphi course is designed to maximise the student's feeling of engagement with the tasks, satisfaction with the work they produce, and sheer enjoyment of the process. During the 12 weeks they spend in the Delphi course they build a variety of games including Horse Race, Memory Match, Noughts and Crosses and even a Delphi version of Space Invaders (cf. Parnaby, 2001). They have an opportunity to experiment with a variety of graphic applications that draw moving shapes, shuffle pictures like a jigsaw puzzle, simulate molecular dispersion and implement Conway's Game of Life. By using the Microsoft™ Agent ActiveX Control (Microsoft™ 2003) students are even able to produce professional quality animation after only a few hours of laboratory work. The result of the "fun" focus of our course materials is that laboratory attendance is essentially 100%, students work happily outside scheduled course times and enthusiasm is high.

However, a fun course that teaches nothing is of little value. Thus, while designing our course material to maximise engagement, we also insure that the programming tasks require the acquisition of necessary skills and concepts, and that the new material introduced in each lesson builds as much as possible on that learned in earlier lessons.

For example, to build the Slot Machine game, students must first come to understand the event-driven interface model. They must learn to place Delphi button, text and image components onto a form, to move between the Delphi screen painter and the programming environment, to manipulate the components via their public methods, to invoke some of Delphi's primitive built-in functions and to actually write the code to make it all go. In the following week's assignment they build upon these skills by meeting one of Delphi's non-visual components: the TTimer. Although some students are puzzled at first by the notion of placing a component that has no visual representation at runtime onto a Form, they quickly realise that "talking to" a TTimer follows exactly the same principles as "talking to" a TButton: one simply invokes its public methods and properties.

Students will learn most comfortably if new skills build seamlessly upon existing skills.

The shift from procedural programming to Object-Oriented programming need not require an abrupt qualitative jump. Once students accept the event-driven programming model (and this is a very natural model for any experienced Mac/OS or Windows user) it is easy to introduce them to Delphi components such as buttons. They are very comfortable writing OnClick handlers for buttons using the Pascal syntax and control structures with which they are already familiar. The only novel syntactic construct they must acquire is dot notation, and they are happy to view this as similar to the syntax for field references when using records. A bit of anthropomorphism ("To tell this TButton to change its caption, you simply say MyButton.caption :=….") is all that is required. Thus,

with only a brief introduction, our students can produce working Delphi applications.

Through the use of Delphi components we introduce our students gradually to some of the basic concepts of Object-Oriented programming such as encapsulation, information hiding, and the communication of two objects via message passing. We do this without new vocabulary or intimidating theoretical discussions. It is accomplished by having students change the colours of TShapes without worrying about the details of how they are drawn on the screen, by having them control the behaviour of the TTimer without understanding exactly how it knows when to fire, by having a button send a message to an Microsoft ™Agent character to produce a complicated animated sequence when they themselves can't draw a good stick figure. They understand that these components distinguish between their interfaces and their implementations, that the latter are hidden from us and we communicate via the former, that each object has certain behaviours and certain responsibilities in the applications that they build. When, at a later point in the course we do begin to discuss the various formalisms of the Object-Oriented method, students should recognise it as simply an extension of the concepts they have already become comfortable using in their programming assignments. Thus we see the Delphi course as providing an easily traversed bridge between their early procedural training and the conceptual understanding required for more advanced Object-Oriented programming, analysis and design.

3. STRUCTURE OF THE DELPHI COURSE

The Delphi course runs over a period of 11 weeks during the second two-thirds of each semester. Each week the course comprises one one-hour lecture and two two-hour labs. The lectures are reserved for the introduction and discussion of theoretical material. The labs are completely hands-on, in keeping with the Otago Polytechnic's general highly applied educational philosophy. Laboratory sessions contain a maximum of 17 students; sufficient student streams are scheduled to maintain this maximum class size. The labs for each stream are taken by a single instructor throughout the term.

For each of the two laboratory sessions, the students receive a detailed handout which presents new content material and contains a series of programming tasks. The first of the two laboratory sessions each week is run as a "group programming" exercise where new material is discussed, and the students and tutor work through the programming tasks together. The second lab each week is comprised of one or two larger programming tasks, and includes minimal new material. Students perform these tasks on their own, with instructor support. These tasks generally include a number of "optional extensions" to challenge the more able students, while still permitting the average student to achieve the satisfaction of a complete programming project (cf Lister & Leaney, 2003).

The first two weeks of the course are primarily devoted to familiarising the students with Delphi and building GUI, event-driven applications. The next four weeks are spent extending their general programming skills while continuing to practice using various complex Delphi components, and emphasising the interface/implementation distinction that allows us to use the complex entities. The behavioural complexity of the components used extends from the TLabel, which simply displays text at a user-determined size and font, all the way to the Microsoft™Agent, which provides complex animation sequences, text-to-speech translation and complex object synchronisation. The tasks involving Agents are among the most popular with students; some of them are detailed in the next section. In the sixth week of the course we introduce the Object-Oriented formalism, and reveal to the students that they have, in fact been doing a bit of Object-Oriented programming all along. In the eighth week students launch into full Object-Oriented coding, by creating their own classes. We will discuss their performance below.

4. EXAMPLES OF TASKS USING AGENTS

We want the programming tasks we assign our students to simultaneously consolidate their basic programming skills, familiarise them with the use of abstract data classes, entertain them and provide a sense of satisfaction in the production of a software product that actually does something. All of Delphi's components provide these features to some degree, but the most dramatic success in our experience is obtained using Microsoft Agents.

Microsoft Agents are animated characters that provide text-to-speech translation and complex animation sequences through a relatively simple set of interface commands. Their behaviours can be managed in a Delphi project using standard flow of control techniques, and they interface easily with other Delphi components. They are also a lot of fun.

We introduce the Agents very early – during the third laboratory session of the course – in the context

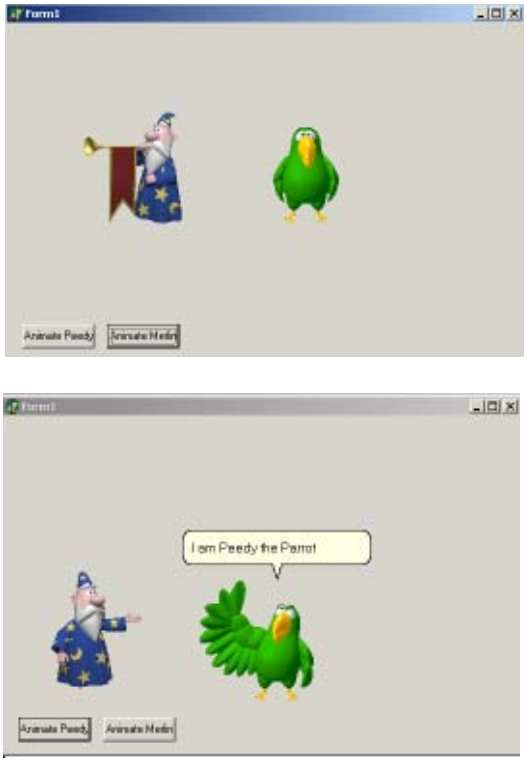

Figure 2. Example Screens From First Agents Task

of a discussion of Information Hiding. Agents are presented as a component with an extremely complex implementation and a comparatively simple interface.

Students first learn how to install the Agent ActiveX control into Delphi. They are taught to place an Agent component onto their Form, just as they would place a button onto their Form. They declare Agent variables (variables of type IAgentCtlCharacter), and recognise that this is logically equivalent to declaring a variable of type integer or type string. They are shown how to initialise the component and assign Agent characters to Agent variables. They are given the syntax of the Agent's *speak* and *play* methods, and a list of the arguments that can be passed to the *play* method (each corresponds to an animation clip). They are then encouraged to experiment with these methods. Two typical screenshots are shown in Figure 2 (sample code is shown in Appendix A - pg108).

Students perform this first Agents task with great enthusiasm, and with a 100% success rate. In the lecture given later that week, we discuss the principles of abstraction and information hiding, and emphasise how nice it is to be able to use a complex data object like Agents, without having to manage the details of their behaviour.

As the course progresses we revisit Agents at various points. Agents are added to the Slot Machine game to give the user feedback about his result and winnings; Agents are used to demonstrate the use of Menus; Agents are used to demonstrate the synchronisation of component behaviour via message passing.

At week six when we introduce classes, objects, encapsulation, responsibility and the other formalisms of the Object-Oriented paradigm, they can be easily explained in terms of the now familiar Agent component. Students understand that Peedy and Merlin are objects, examples of the class Agents. They understand that all Agents have behaviours like speak and play, because they have often instructed Peedy and Merlin to perform those behaviours and observed their actions in response. They understand that the responsibility for implementing these behaviours lies with the object; the user need only invoke the interface. Thus students recognise objects and classes as familiar entities, not as something new and alarming.

5. HOW DO WE DEFINE "GOOD PROGRAMMER"?

We naturally hope that our approach to the teaching of Object-Oriented logic will allow students to move without distress from the procedural paradigm to the Object-Oriented paradigm. We want them to be "good Object-Oriented programmers". Unfortunately, there seems to be no clear metric for what constitutes a "good Object-Oriented programmer"; there are precious few metrics for what constitutes a good programmer at all (e.g. Barr, *et. al.,* 1999; and see Fincher, 1999). Do we want them to write efficient code or readable code? Do we care most about robust bug-free code, or are we concerned more with the elegance of the algorithm? Does it matter if students' Object-Oriented programming is not entirely pure, as long as it meets all the functional requirements for the task? These questions are difficult enough to answer without even addressing the fact that terms such as "readable" and "elegant" are without any clearly defined operational definition. What data are we to collect and analyse when we wish to measure our student's programming skill?

In the literature, authors tend to use expressions like "the students seem to understand the concept" or "most students seemed confused by the task". Occasionally authors provide the results of student self-reports of the degree to which they found the course presentation "useful", or "confusing" or "interesting". None of these approaches is specific enough to allow us to compare in a rigorous fashion

the success or failure of a given programming course in producing "good programmers".

We see the need for developing a satisfactory metric for programming ability as being essential to the sensible comparison of pedagogical approaches, and thus to the development of a truly effective curriculum. However the lack of such a metric in the literature highlights the difficulties involved in its development. We consider this an exciting area for further research.

6. DETERMINING THE EFFECTIVENESS OF OUR DELPHI BRIDGE

To analyse the success or otherwise of our new Delphi course, we intend in the first instance to adapt the technique of Robins *et. al.* (2001). Robins *et. al.* have defined a list of 36 types of errors seen in their introductory Java programming course at the University of Otago. These errors are grouped into three categories: problems with basic tools (i.e. trouble with the authoring environment itself); problems with programming logic; and problems with the programming language. In assessing the performance of their own students, they observed them at work on a variety of programming tasks, and studied the pattern of errors made on each task. The classification they define gives good coverage, and includes all the errors that programming instructors commonly see their students make.

For our purposes, we have simplified the checklist and modified it for Delphi. During our students' first two laboratories on Object-Oriented programming, we observed their performance, and classified the errors they made. Whereas Robins et. al. recorded student difficulties only when a student specifically requested help, our smaller class size allowed us to monitor each student's performance throughout the task. We thus observed both problems that students were able to resolve themselves, and problems that required tutor support.

Students had been achieving near 100% success on the laboratory session tasks to this point in the class, and since we believed the first Object-Oriented task to be of a comparable difficulty, we would have to interpret any significant failure rate with the first Object-Oriented task as an indication that our "gentle bridging" approach was not entirely successful. Alternatively, if students were comfortably able to complete their first Object-Oriented task, we could conclude that our combination of core skill training, gradual introduction of concepts and elevated engagement had served to move students smoothly from the procedural to the Object-Oriented paradigm.

7. STUDENT PERFORMANCE

With our small class sizes, and with only two laboratory sessions available for student observation, we were able to collect only a very limited amount of data; future presentations of the course will allow us to continue the observation process. However, even with such a small sample, patterns of student performance were apparent, and as a reflection of the success or failure of our approach, the results are mixed. Students were unable to achieve the 100% completion rate that we hoped for without significant tutor intervention, but the errors they made were systematic, and in one case seem to indicate a confusion more with the syntax of Object Pascal than with the core concepts of the Object-Oriented paradigm. Since predictable misunderstandings are the easiest to resolve, and task completion was very high with tutor support, we find these results encouraging.

7.1 Grasp of Core Concepts

Students were extremely comfortable with the notion of defining classes. All students were able to respond correctly to questions such as "What properties should class TLibraryBook have?" and "What methods do we need for THorseRacer"? All students were able to correctly declare their classes in Delphi, using the **private** and **public** keywords as appropriate to distinguish between fields and methods. Further, when provided with a prewritten class, students were all able to correctly declare class instances as variables, and to correctly invoke their methods using dot notation.

7.2 Confusion of Class Declaration and Class Instance

The student's main difficulty arose in the implementation of methods. For example, students were instructed to create a class TSquare, that had properties sideLength, Colour, XCoordinate and YCoordinate, and methods SetMySideLength, SetMyColour and DrawMe. We provided students with an OnClick handler that declared a variable Square1 of type TSquare, and set its SideLength property via the call: Square1.SetMySideLength(50).

Unfortunately, when we asked students to implement the SetMySideLength method, they invariably wrote:

```
Procedure
 TSquare.SetMySideLength(n:integer);
Begin
  Square1.SideLength := n;
End;
```

That is, they referred to a *class instance* inside the definition of the method. After several weeks of Delphi programming, where all properties and methods are referred to using this dot notation, students were extremely reluctant to simply (and correctly) write:

```
Procedure
 TSquare.SetMySideLength(n:integer);
Begin
  SideLength := n;
End;
```

In one laboratory class of 11 students, every single student made this error. A common student question was "How does it know whose SideLength property to use?" When the tutor explained that the system would use the SideLength property of the invoking object, the student would exclaim and nod, and, invariably, proceed to make exactly the same error when defining the next method.

We are currently running our third Object-Oriented laboratory session, and students continue to struggle with this construct. Although they claim to understand the distinction between the class definition and instances of the class, when they write code for a method, they still want to have an instance of the class to refer to. The notion of writing their code for "whichever class instance calls the method" is extremely difficult for them to grasp.

Robins *et al.* call this problem "Class vs. Instance" and observed it with their students, although proportionally less than we do. We suspect that this is because of our students extensive pre-exposure to the dot notation during the procedural weeks of the course. Robins' Java course does not have this procedural component. Perhaps the retroactive inhibition responsible for the traditional paradigm shift problem is not localized to experienced programmers, but to individual constructs that are highly familiar. In future presentations of the course we will need to explore ways of avoiding this problem. Our first instinct is to emphasize the difference between class definition and class instance from the introduction of our first simple Delphi component. It may be helpful to show students a simplified version of some of the methods for buttons and labels, demonstrating that they contain no instance reference.

7.3 Trouble with a Syntactic Anomaly

The second common error made in these early exercises also seems to reflect student's pre-exposure to Delphi syntax. The ordinary method call in Delphi has the form *ObjectVariable.MethodName*. The exception to this is the **create** method, where the syntax is *ObjectVariable := ClassName.Create.* In their first programming tasks, students consistently attempt to call *ObjectVariable.Create*. We view this as actually quite sensible, and the required syntax as an anomaly. It does, however, demonstrate again the impact of student's prior experience using the dot notation to reference preexisting Delphi objects.

7.4 Facility with Basic Coding

We were pleased to observe that, when the errors described above were resolved, our students were able to implement relatively complex behaviors involving branching, looping and traversal of arrays within their class methods. Thus our emphasis on students' developing good basic Pascal skills has facilitated this aspect of their Object Oriented development. We are optimistic that once they become more experienced with "thinking Object-Oriented" they will be able to produce elegant and robust Object-Oriented applications.

8. SUMMARY

Tertiary institutions around the world are struggling to find the optimal technique for producing programmers who can write commercial quality code using a variety of tools and paradigms. The course we are now developing at Otago Polytechnic hopes to instill strong programming skills and an awareness of the variety of possible programming paradigms in the first year.

Our first experience with the course has demonstrated that it is possible to design course materials that are engaging and tractable for students who have only a limited amount of programming experience. Use of advanced Delphi software components serves to consolidate mechanical programming skill while simultaneously introducing students to the concepts of encapsulation and inheritance without the complexities of a new formalism. When that formalism is later introduced, students accept the concepts with ease due to their familiarity.

Unfortunately, the mechanics of implementing these concepts still flounders on the rocks of students' prior habits with the language. Students seem to learn to

use complex components without gaining an understanding of the distinction between the generic class and the specific object instance. We intend to explore ways of modifying the course to address this problem.

Integral to the question of whether a course is effectively producing good programmers is the need to be able to quantify "good programming". It is not possible to compare accurately different pedagogical methodologies without having some metric for measuring student's programming skills. This question is somewhat neglected in the literature, with only a few attempts being made at developing a precise metric. We consider the development of such a metric to be essential, and plan to devote further research efforts to it. It is important that such a metric measure programming skill in the early stages, when expertise cannot be expected. An accurate technique for understanding early programming performance is necessary before we can build a first course that consistently starts our students on the long road to becoming Good Programmers.

REFERENCES

Barr,M., Holden,S., Phillips,., Greening,T. (1999) "An Exploration of Novice Programming Errors in an Object-Oriented Environment", SIGSCE Bulletin, 31,4:42-46.

Bergin, J. (2000) "Why Procedural is the Wrong First Paradigm if OOP is the Goal" , http://csis.pace.edu/~bergin/papers/WhyNotProceduralFirst.html

Decker, R. and Hirshfield, S. (1993) "Top-Down Teaching: Object-Oriented Programming in CS1", ACM 24th CSE, 1993.

Duke, R.. Salzman, E., Burmeister, J., Poon, J., Murray,L., (2000) "Teaching Programming To Beginners – Choosing The Language Is Just The First Step", Proceedings of the Australasian Conference on Computing Education, pp. 79-86, December 2000, Melbourne Australia.

Fincher, S. (1999) "What are we doing when we teach programming?", 29TH ASEE/IEEE Frontiers in Education Conference, November 10-13, 1999, San Juan, Puerto Rico.

Guzdial, M. (1995) "Centralized Mindset: A Student Problem with Object-Oriented Programming.", SIGCSE'95, 3/95 Nashville TN.

Guzdial, M. and Soloway, E., (2000) "Teaching the Nintendo Generation to Program", Communications of the ACM, 45,(4): 17-21.

Guzdial, M., (1995) "Centralized Midnset: A Student Problem with Object-Oriented Programming", SIGSCE '95, Nashville TN.

Kaasboll, J. (2000) "Learning and Teaching Programming", Lecture at Interface 2000, University of Pretoria, 19-20 May, 2000.

Kearsley, G. & Sheniderman, B. (1999) "Engagement Theory: A Framework For Technology-Based Teaching And Learning" http://home.sprynet.com/~gkearsley/engage.htm

Lister, R., & Leaney, J. (2003) "Introductory Programming, Criterion-Referencing and Bloom", SIGCSE'03, February 19-23, Reno, Nevada.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001) "A multinational, multi-institutional study of assessment of programming skills of first-year CS students", ACM SIGCSE Bulletin, 33(4):125-140.

Microsoft ™ (2003) http://www.microsoft.com/msagent/default.asp

Nelson, H.J., Armstrong, D.J., & Ghods, M. (2002) "Old Dogs and New Tricks", Communications of the ACM, 45,(10):132–137.

Parnaby, O. (2001) "A Space Invaders Clone in Delphi", http://www.sebas.vic.edu.au/staff/oparnaby/delphi/spaceinvaders/spaceinvaders.htm

Robins, A., Rountree, N., Rountree, J. (2001) "My Program Is Correct, But It Doesn't Run: A Review Of Novice Programming And A Study Of An Introductory Programming Paper", University of Otago Technical Report, OUCS-2001-06.

Stein, Lynn Andrea (1998) "What We've Swept Under the Rug: Radically Rethinking CS1.", Computer Science Education 8(2):118-129.

Webster, J & Hayes, H. (1997) "Audience Engagement in Multimedia Presentations", ACM SIGMIS Database, 28,(2):63-77.

Appendix containing sample code on page 108