Teaching introductory programming in the multi-media world

Ursula Wolz, Scott Weisgarber, Daniel Domen and Michael McAuliffe
Department of Computer Science
Trenton State College
Hillwood Lakes, Box 4700
Trenton, NJ 08650
wolz@trenton.edu

Abstract

We address the problems of effectively teaching introductory programming in the multi-media age. We provide a short history on user interfaces, contrasting the line oriented, turn taking dialogue model with the event driven, desktop model. We summarize the major conceptual outcomes of this shift: the event model itself, the object-oriented paradigm, and the more detailed classification of I/O types. We describe how the current generation of tools rely on programmer convention, thus encouraging sloppy coding. We constrast three current approaches to teaching programming with multi-media and present an approach that exploits the advantages of all three.

1 Introduction

Historically, designers of programming languages have given short shrift to concerns of input output (I/O). The rational was that I/O was machine dependent and therefore beyond the scope of establishing language standards. This has always been the bane of teachers of introductory computer science. One cannot create meaningful exercises without including instruction in I/O, yet teaching I/O techniques can often degenerate into discussions of technical minutia that are only marginally relevant to the larger goals of the class. The dilemma is that I/O programming can be profoundly rewarding. It is satisfying to a novice programmer to produce a glossy visual output no matter how awful the underlying algorithm might be. Multi-media and graphical user interfaces (MM/GUI) have not alleviated this problem. In fact, we claim they have made the problem worse because it is harder than ever to control I/O without mastery of esoteric minutia.

This paper provides an overview of the problems of effectively teaching introductory programming in the multi-media age. Our discussion summarizes the changes over the past two decades[1, 2, 3, 4, 5, 6]. Our perspective is unique however, both in its historical analysis and its approach to solving the problem. We provide a short history on the evolution of user interfaces, contrasting the old line oriented, turn taking dialogue model with the event driven, desktop model currently available.

Integrating Tech. into C.S.E. 6/96 Barcelona, Spain
© 1996 ACM 0-89791-844-4/96/0009...$3.50

We summarize the major conceptual outcomes of this shift: the event model itself, a practical use for the object-oriented paradigm, and the more detailed classification of I/O types. We claim the current generation of tools that support I/O programming suffer from a classic problem: they do not provide clean mechanisms that impose discipline on the programmer. Instead programming is done by *convention* which, when violated, still may produce working code. Environments that rely on convention unintentionally encourage sloppy programming.

We intend to show that this is not just a technology problem, but that it is a pedagogical problem as well. We will describe current approaches to teaching programming with MM/GUI and discuss their advantages and disadvantages. We will also show how the tools themselves encourage a sequence of instruction that seductively re-enforces bad rather than good programming habits. Finally we will describe how we are attempting to attack this problem both pedagogically and technically: by creating useful cross-platform tools that have a direct mapping to global concepts and by sequencing instruction to avoid encouraging bad habits. The remainder of this paper elaborates on these ideas.

2 The evolution of interfaces

User interface technology can be crudely divided between the old teletype, line based technology, and graphical user interfaces. We stress that this is a gross simplification for the purposes of this paper. In particular we ignore the intermediary stages of text/graphics overlays of the early personal computers. The old model was a text-based turn-taking dialogue between a user and the program[6]. The program provided a prompt with a restricted set of choices that in its most elaborate form became a hierarchical menu. The user's response would result either in the desired effect, a request for more information or a complaint that the user's actions were not allowed. The new GUI/MM model is event driven[4,5]. The user and the program can initiate events that result in various kinds of displays or prompts, as well as the execution of data crunching algorithms. The turn-taking dialogue has been replaced by a more open-ended interaction in which the user can change the focus of the interaction. This is the most significant change from the user's standpoint. In the turn taking model, the dialogue was initiated and controlled by the program, in the event driven model the user initiates the events that produce graphical responses from the program.

The old model was relatively easy to learn to program. Typically the programming language of choice had a few simple function calls that controlled the 'getting' and 'putting' of streams of text characters. Programming the new model is entirely dependent upon the language of choice. Some environments, most notably Hypertalk[2] exploit a GUI to allow programmers to define interfaces for their own program. The disadvantage of these languages is that they tend to be restrictive, limiting the programmer's ability to create efficient code. Environments that support C or Pascal programming provide greater flexibility for the programmer, but require mastery of a significant amount of minutia, both in terms of syntactic form and knowledge of relevant libraries[5].

A major distinction between the old model and the new is the shift from a procedural/functional paradigm to an object-based one. Regardless of whether the environment explicitly supports true object-oriented programming, all GUI/MMs we have examined rely implicitly on an object model to support data encapsulation and polymorphism regardless of whether the underlying programming language claims to be object oriented. Borland C for the PC compatibles[1] and XView for Unix[3] environments are prime examples.

We claim that the reliance on objects without explicit object mechanisms is part of the reason GUI/MM programming is so hard for beginners. The environment requires that the programmer construct code using a discipline that is merely a convention. It is not imposed by the environment. To clarify this consider the original BASIC environments that only supported global variables. Good programmers used *conventions* for creating local environments within subroutines. Similarly when programming GUI/MM using general languages like C and Pascal, one must pay careful attention to elaborate conventions, such as calling badly named function as part of an initialization and including obscure pointer references within calls. More than one manual we studied simply said in effect 'do this, but don't ask why yet.'

The final difference between the old and new models is the classification of I/O types which are used to focus and constrain the interaction. This is a clear advantage over the old model. In text-based interfaces the programmer built interactivity using a suite of simple text stream primitives, typically two for fetching and displaying a single character and two for fetching and displaying a line of text. In the GUI/MM model, input in particular is classified by the type of object that gathers a particular type of data[4]. For example graphics and text windows allow users to enter unrestricted data. Dialogue boxes constrain the user's input in order to tailor it to a response. Sliders provide a clean mechanism for specifying a point within a range. Buttons provide an extremely constrained set of options. This variety has always been necessary, and in fact in the text-based environments, user error was often introduced precisely because the response was not properly constrained. The problem with the new classifications is that they are typically defined in terms of the tools themselves (buttons vs. sliders) rather than in terms of their impact on the dialogue. The danger is that novices learn what the tools are, but never come to appreciate their use. For example, a task that could be better served by a button is done through a dialogue box.

3 The pedagogical problem

The teacher of introductory programming in a multi-media environment must address three issues within the larger agenda of teaching programming principles: (1) how to teach the event model, (2) how to encourage adherence to convention when it is not imposed, (3) how to instill appreciation for the different types of GUI/MM tools that are available. Currently there are three approaches in an already overburdened curriculum: tackle it head on, begin with a 'user friendly' language, or avoid GUI/MM programming altogether.

The advantage of tackling the problem head on is that students do something meaningful and interesting. They learn real skills and create visually satisfying programs. The disadvantage is that learning in environments that rely on convention (such as C), requires careful attention to detail. This can be frustrating for beginners who are trying to master both concepts and style simultaneously. The second disadvantage is that the course can become focused on the interface rather than the more general issues of programming. Although students produce glossy results by specifying a collection of devices, they may never write a meaningful algorithm.

A second approach is to avoid I/O programming, giving assignments that include only minimal need for it. This is most easily accomplished in interpreted languages such as Scheme. The advantage is that the focus can be on the big issues such as algorithm design, modularity, expressibility, and data description. The little I/O that is done is accomplished via standard drivers, for example, simply listing output or using a standard old fashioned input prompt. One disadvantage is that a major theme, namely code robustness is often violated. For example to prevent a program crash, numeric input should be read as a string and then converted. It is a delicate problem for the instructor to determine whether the data conversion problem is relevant in the larger context of the particular class. The other disadvantage is that the programs students write are often boring by current standards because the power of producing interesting visual output is lost.

A third approach is to have students develop code fragments that are inserted in a 'wrapper' provided to them. The advantage of assigning code fragments is that students see how their contribution fits into a bigger picture. They can produce visually appealing results. If they are motivated they can study the surrounding code and model from it. The disadvantage is that many students will not study the surrounding code, missing out on developing an important skill. Furthermore they may never take off the training wheels and avoid the inevitable when they must write a complex program from scratch or near scratch. Finally, this approach has the potential for providing students with a sense of mastery over interfaces that they barely control. At some point they must be explicitly taught interface concepts, including the event model and the new I/O types. They must learn to make real decisions about what kind of I/O techniques to apply in a given situation.

4 Is there a solution?

The issues highlighted above indicate that the solution lies both in the technology and the teaching. One could hope that the perfect language will be developed, but history indicates that the balance between accessibility for novices and extensibility for advanced programmers may be impossible to achieve in a single environment. Although introductory programming may be taught in a 'friendly' language, inevitably computer science majors must tackle the hard and powerful environments currently exemplified by C/C++. MM/GUI merely compounds an old problem because the interface has become less accessible to the programmer. While some glossy results may be easier to achieve (running a video for example) their pedagogical worth may be questionable.

Our solution is three fold and occurs over two semesters. We use two programming environments and pay careful attention to the teaching of concepts over techniques. We also use a set of in house interface tools as well as simple code wrappers to reduce the complexity of the students' initial exposure to interfaces. We begin by teaching in a 'safe' interpreted environment. Currently we use Mathematica. Explicit graphics programming is required of our students only when it directly re-enforces a major programming concept. For example, students write functions to plot the classic time complexity functions, n, n^2, $n\log_2 n$, or to solve a maze using a stack, and display their solution graphically. We teach basic interface concepts within the safe environment. We then move into a C/C++ environment, introducing object concepts first in the abstract and then as part of our introduction to GUI. The event model is taught as an obvious application of iteration constructs.

In-house libraries provide initial isolation from detail. We currently support programming within Macintosh, Windows and Xterminal interface paradigms. In all instances we also provide students with supplemental material that explains how the libraries work. This material is targeted at the better students and is not required for the course.

Our approach is not unique, however we carefully and deliberately address the basic problems outlined here, diminishing the disadvantages of the approaches described and increasing their individual advantages by selecting the best of each. In particular we explicitly talk about convention in our classes. Students are given examples of proper convention but are also shown how convention can be violated, and what the consequences of cutting corners can be.

It is too soon to provide definitive data on how our students master the basic concepts of programming. Anecdotal evidence suggests they emerge from our experience with less detailed technical expertise in a particular language, but with a greater flexibility to approach new languages, paradigms and interface technologies. Our use of multi-media is intentionally conservative. Our goal remains to motivate students to problem solve from a modular perspective. They will have plenty of time later in the curriculum to develop a multi-media bag of tricks.

References

1 *Borland C Users Guide.* Scotts Valley, CA: Borland International, 1991.

2 Decker, R., Hirshfield S. *The Analytical Engine, An Introduction to Computer Science Using HyperCard 2.1.* Boston, MA: PWS Publishing 1994.

3 Heller D., *Xview Programming Manual.* Sebastopol, CA: O'Reilly & Associates, 1991.

4 Shneiderman, B. *Designing the User Interface Strategies for Effective Human-Computer Interaction.* Reading, MA: Addison Wesley, 1992.

5 Sydow, D. *Foundations of Mac Programming.* Foster City, CA: IDG Books, 1995.

6 Wirth, N. *Algorithms + Data structures = Programs.* Englewood Cliffs, NJ: Prentice Hall, 1976.