

# Tools for Teaching Introductory Programming: What Works?

Kris Powers

Paul Gross

(Moderators)

Tufts University

Medford, MA 02155

kpowers@cs.tufts.edu

pgross01@cs.tufts.edu

Steve Cooper

Saint Joseph's University

Philadelphia, PA 19131

scooper@sju.edu

Myles McNally

Alma College

Alma, MI 48801

mcnally@alma.edu

Kenneth J. Goldman

Washington University

St. Louis, MO 63130

kjg@cse.wustl.edu

Viera Proulx

Northeastern University

Boston, MA 02115

vkp@ccs.neu.edu

Martin Carlisle

Air Force Academy

USAFA, CO

carlisle@acm.org

## Categories and Subject Descriptors

K.3.2 [Computer and Information Science Education]: Computer Science Education

## General Terms

Experimentation, Human Factors

## Keywords

Novice programming environments, introductory programming

## 1. SUMMARY

In the past decade educators have developed a myriad of tools to help novices learn to program. Different tools emerge as new features or combinations of features are employed. In this panel we consider the features of recent tools that have garnered significant interest in the computer science education community. These including *narrative tools* which support programming to tell a story (e.g., Alice [6], Jeroo [8]), *visual programming tools* which support the construction of programs through a drag-and-drop interface (e.g., JPie [3], Alice [6], Karel Universe), *flow-model tools* (e.g., Raptor [1], Iconic Programmer [2], VisualLogic) which construct programs through connecting program elements to represent order of computation, *specialized output realizations* (e.g., Lego Mindstorms [5], JES [7]) that provide execution feedback in non-textual ways, like multimedia or kinesthetic robotics, and *tiered language tools* (e.g., ProfessorJ [4], RoboLab) in which novices can use more sophisticated versions of a language as their expertise develops.

Each panelist has significant experience with a tool in which one of the listed features is of predominant importance. We ask each to address why the feature they represent is effective in supporting novices, and what other features or combinations of features they feel are most important to making novice programming environments as effective as possible. Our hope is that such discussion may help guide future development.

## 2. PANEL STATEMENTS

**Steve Cooper (narrative tools):** Alice is a 3-dimensional interactive animation environment used for introducing novices to object-

oriented programming. The specific Alice feature being presented in this panel is the use of narrative. The two types of virtual worlds students build are interactive worlds (such as computer games) and non-interactive worlds (movies or stories).

Storytelling has two strengths. The first strength is that we have found that the ability to direct your own movie is extraordinarily attractive to a wide range of students. Alice's storytelling has particular appeal to female students, many of whom refer to "programming with a purpose" as a particularly attractive feature. Many minority students also appreciate the aspect of storytelling, as many come from cultures where storytelling/oral tradition plays an important role. The second strength is that the construction of programs as story readily allows us to introduce storyboarding as a design technique; a technique that appeals to students, as they are already familiar with the concept of storyboarding in the making of animated motion pictures.

The main appeal of storytelling is in attracting students to be interested in the programs they construct. Motivation is important, and it leads to students spending significant time on task.

**Ken Goldman (visual programming tools):** Introductory students can be distracted learning programming language syntax and how to interpret compiler error messages. To address this problem, some novice programming environments provide a visual interface in which programs are constructed using gestures, such as drag and drop. A general philosophy of these environments is that *the user interface enforces program structure*, preventing syntax errors. With each gesture, the user takes the program from one legal state to another. Additionally, this visual approach elevates the unit of discourse from the character to the semantic unit by allowing users to *directly manipulate graphical representations of programming concepts* rather than type individual characters.

By enforcing program structure, the program is always in an executable state. JPie, which provides visual construction of Java programs, exploits this property to support *live development of running programs*. Changes take effect immediately on existing instances, tightening the feedback cycle. With immediate feedback, we have seen students understand concepts more and our course can cover more concepts without losing valuable hands-on experience.

A well-known problem with structured editors is that they can be rigid since each edit of a program must leave it in a legal state. To address this problem, JPie provides a *relaxed edit-time grammar* which allows four types of discrepancies that leave the program in

Copyright is held by the author/owner(s).

SIGCSE'06, March 1-5, 2006, Houston, Texas, USA.

ACM 1-59593-259-3/06/0003.

an unexecutable intermediate state. By allowing a few carefully chosen discrepancies, the visual environment can offer editing flexibility that is much closer to that provided by textual environments while still lowering the syntax burden.

**Martin Carlisle (flow-model tools):** Visual programs based on flowcharts allow students to visualize how programs work and develop algorithms in a more intuitive fashion. We have learned that, given a choice, over 95% of our students selected to use a flowchart to represent an algorithm, even when they had seen flowcharts in only a single lecture and had been taught in 3<sup>rd</sup> generation programming language. The flow model greatly reduces syntactic complexity, allowing students to focus on solving the problem instead of finding missing semicolons.

To provide interesting problems for students to solve, the environment needs to provide a rich set of graphical capabilities as well as numbers, strings and arrays. It would be interesting to see a combination of the 3D graphics of Alice with the visualization of control flow provided by RAPTOR.

Of course, the end goal is student learning and performance. Experiments on introductory programming environments need to demonstrate with hard numbers that student learning has increased. We were pleased to see scores on the problem solving section of our final exam increase with statistical significance exceeding 95%.

**Myles McNally (specialized realization):** Lego MindStorms, the inexpensive robotics kit from LEGO, has been used by a variety of CS educators in a wide range of courses. Programming of the robots can be accomplished through numerous languages, including Java and C++, and through visual programming environments such as Robolab.

So why employ MindStorms in teaching? For many educators the first appeal is the motivation factor. Most students played with LEGOs as a child, and find the idea of working with LEGO-based robots appealing and the actual experience enjoyable. This leads to increased time on task, which has been shown in numerous studies to be a key factor in positive learning outcomes. In addition, the lab experience itself is improved. Working with a physical artifact such as a MindStorms robot breaks up the usual CS lab experience (sitting in front of a computer screen and typing) into an active-learning session where students code and test their solutions in a public way, leading to increased interactions with fellow students.

More importantly, MindStorms robots literally embody state and behavior, physically modeling the structure of the programming solutions. Their activities are the concrete instantiation of program behavior. Students receive immediate visual feedback, allowing visual debugging since the program's state of execution is literally played out in front of them.

**Viera Proulx (tiered language tools):** DrScheme programming environment supports the whole spectrum of programmers - from complete novices to world class researchers and real world applications. For novice programmers there are two environments: How to Design Programs series of Scheme-like languages and ProfessorJ series of Java-like languages. The teaching languages start with the minimal syntax and complexity, with error messages designed to provide feedback appropriate for the students' understanding of the language. As the course progresses, new features are added and new error messages are modified to reflect

students' current understanding of the language. Both language series delay the introduction of mutation and assignment until the advance language levels.

Starting with a simple syntax of beginner level language students can focus on the design of the program and understand clearly the most salient points of program design. Additional language features are added when programming in the simple language becomes tedious and repetitious. Students recognize the need for a more complex feature - indeed in our experience they outright beg for it. This provides the opportunity to explain a formal process for designing abstractions. An additional advantage is that students write complete programs from the beginning with all design steps in place - at a simple level: problem analysis, documentation, test design and the evaluation of test results.

It is important to make sure that all parts of the environment - the language, the error messages, the testing environment, and any supporting libraries, all respect the current language constraints. For example, in DrJava student tests are written as JUnit classes and must be part of a method that returns void, even though the beginner language does not allow such methods, thus introducing a dichotomy.

### 3. REFERENCES

- [1] Carlisle, M. C., Wilson, T. A., Humphries, J. W., and Hadfield, S. M. 2005. RAPTOR: a visual programming environment for teaching algorithmic problem solving. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 176-180.
- [2] Chen, S. and Morris, S. 2005. Iconic programming for flowcharts, java, turing, etc. In *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education*. ACM Press, 104-107.
- [3] Goldman, K. J. 2004. A concepts-first introduction to computer science. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 432-436.
- [4] Gray, K. E. and Flatt, M. 2003. ProfessorJ: a gradual introduction to Java through language levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 170-177.
- [5] Kay, J. S. 2003. Teaching robotics from a computer science perspective. *J. Computing in Small Colleges*. 19, 2 (Dec. 2003), 329-336.
- [6] Moskal, B., Lurie, D., and Cooper, S. 2004. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 75-79.
- [7] Rich, L., Perry, H., and Guzdial, M. 2004. A CS1 course designed to address interests of women. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 190-194.
- [8] Sanders, D. and Dorn, B. 2003. Classroom experience with Jeroo. *J. Comput. Small Coll.* 18, 4 (Apr. 2003), 308-316.