RAPTOR: A VISUAL PROGRAMMING ENVIRONMENT FOR TEACHING OBJECT-ORIENTED PROGRAMMING

Martin C. Carlisle
Department of Computer Science
United States Air Force Academy
carlislem@acm.org

ABSTRACT
Learning object-oriented programming usually involves learning a programming language with a large amount of complexity. Students very often spend more time dealing with syntactical complexity than learning the underlying principles of object-orientation or solving the problem. Additionally, the textual nature of most programming environments works against the learning style of the majority of students. RAPTOR is an iconic programming environment, designed specifically to help students visualize classes and methods and limit syntactic complexity. RAPTOR programs are created visually using a combination of UML and flowcharts. The resulting programs can be executed visually within the environment and converted to Java.

1. INTRODUCTION
"Programming courses often focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details rather than the underlying algorithmic skills…. Many of the languages used for object-oriented programming in industry—particularly C++, but to a certain extent Java as well—are significantly more complex than classical languages. Unless instructors take special care to introduce the material in a way that limits this complexity, such details can easily overwhelm introductory students." [1]
In our experience, even when instructors try to focus on the more fundamental concepts of classes and algorithms, they are forced to spend a significant amount of class time on syntactic difficulties that students encounter.

Furthermore, Felder [2] notes that most students are visual learners and that instructors tend to present information verbally. Between 75% and 83% of students are visual learners [3,4]. Traditional programming languages, textual in nature, provide a non-intuitive framework for learning about object-orientation and algorithmic thinking for the majority of our students. Scanlan [5] showed that students understand algorithms presented as flowcharts better than those presented in pseudocode. Carlisle et. al [6] showed that, when given a choice, 95% of students chose to express algorithms using flowcharts rather than using a traditional programming language, even when the majority of their instruction had been done in a traditional language. Several studies [6,7,8] showed that students performed better in courses when taught with iconic programming languages.

Since there was a large body of evidence supporting the idea that students understand programming concepts better when given a visual representation, we created a visual programming environment for introducing object-oriented programming. RAPTOR allows students to create algorithms by combining basic graphical symbols. Students create their class hierarchy in a UML designer and then represent method bodies as flowcharts. The resulting programs can then be run in the environment, either step-by-step or in continuous play mode. The environment visually displays the location of the currently executing symbol, as well as the contents of all variables. Also, RAPTOR provides a simple graphics library, based on AdaGraph [9]. Not only can the students create algorithms visually, but also the problems they solve can be visual.

We are using RAPTOR in an Introduction to Programming course. The course is primarily taught in Java, and RAPTOR is used to visualize how objects work. Students are able to create their designs in RAPTOR and then convert the result to Java.

2. RELATED WORK
A significant number of visual or iconic programming environments have been developed. SFC (Structured Flow Chart) Editor [10] is a structured flowchart editor by Tia Watts. SFC allows the user to develop a structured flowchart, and always displays a psuedocode representation of the flowchart in either a C++ or Pascal-like syntax. The user then copies and pastes the textual representation into a text editor or integrated development environment (IDE) and makes changes to get a complete program. Although SFC generates C++ code, it uses an imperative subset of the language.

Calloni and Bagert [8] developed a Windows-based iconic programming language, BACCII++. They use BACCII++ as a supplement to C++ in their CS1/CS2 sequence. BACCII++ supports the creation of classes and methods; however, we were unable to find any place to download or purchase the tool.

Visual Logic [11] is a commercial tool (~$31) based on an academic project, FLINT [12]. Visual Logic supports creation of programs with multiple procedures, each of which is represented as a flowchart. The language contains some built-in functions from Visual Basic. As with SFC, Visual Logic does not support the creation of classes.

Alice [13], by Carnegie Mellon University, is a widely used 3D programming environment that supports teaching introductory programming concepts in an object-based way. Students create animations by placing objects in a 3D virtual world, and then programming their behavior. Although Alice uses object terminology, it does not directly support inheritance [14].

Iconic Programmer [15] and B# [7] are two other tools that allow students to create programs using flowcharts. They support input/output, selection, looping, and code generation but do not support subprograms.

RAPTOR is an open-source tool that fully supports object-oriented programming, including encapsulation, inheritance and polymorphism. RAPTOR enables students to execute their algorithms within the environment, rather than having to separately compile and execute their programs. This means that debugging can be done on the visual representation of the algorithm, rather than the textual one and prevents having to use multiple tools. This combination of features makes RAPTOR unique, providing functionality not available with any other currently existing educational programming environment.

3. RAPTOR

RAPTOR is written in a combination of Ada and C#, and runs in the .NET Framework. RAPTOR begins by opening a UML diagram, in which users can create classes, interfaces and enumeration types and specify relationships between them. The UML designer is based on the open source tool NClass by Balazs Tihanyi [16]. An example hierarchy created is shown in Figure 1.

The UML Designer allows users to create classes, interfaces and enumeration types. These can be given the Java access modifiers of public, private, protected or default. Additionally, classes can be specified as abstract, sealed, or static. A zoom bar allows the user to resize the diagram as desired, or make it fit the current window. The UML diagram can also be annotated with comments. Each of these UML elements can be moved on the diagram.

The UML window also allows for the specification of relationships between entities. Possible relationships are inheritance, interface implementation, class nesting, association, composition, aggregation and dependency. As with the elements, the arrows indicating the relationships can be moved on the diagram.

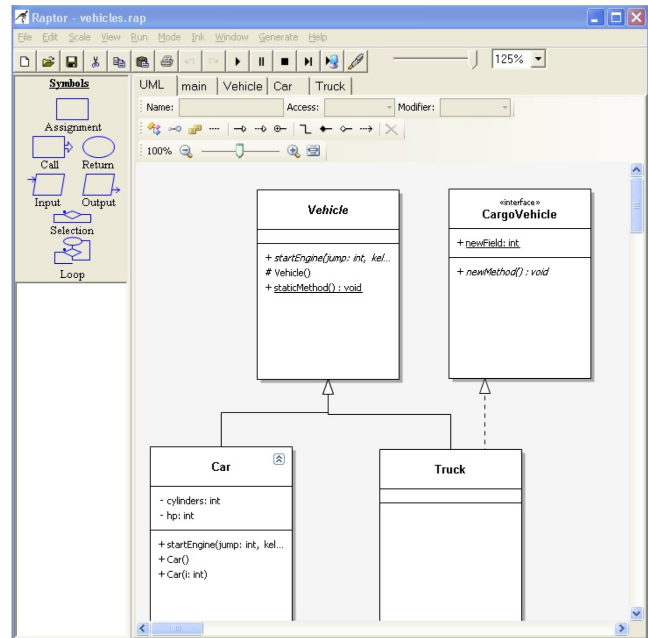

Figure 1: RAPTOR UML Designer.

Once a class has been created, users can add methods and attributes by double-clicking on it. This brings up the class editor (see Figure 2).

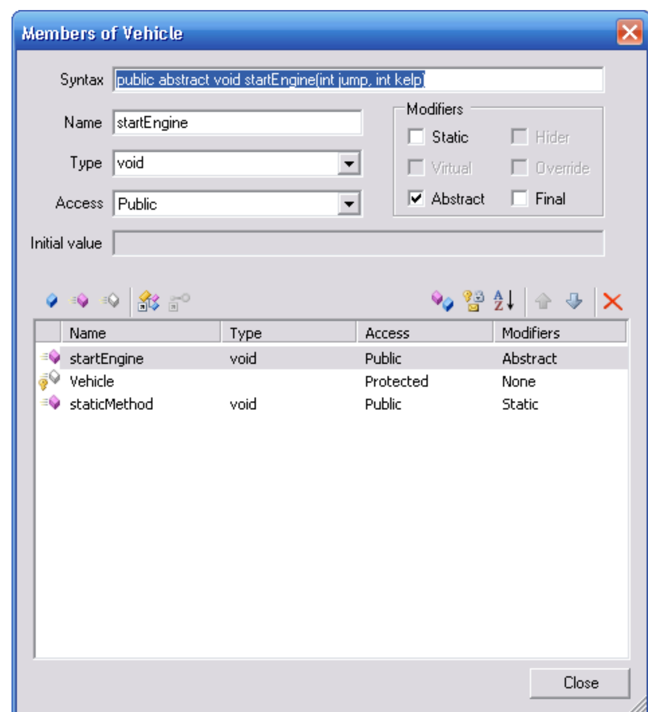

Figure 2: RAPTOR Class Editor.

The class editor provides a mechanism for directly editing the Java syntax of the method, attribute, or

constructor. It also provides helpful GUI tools for students unfamiliar with this syntax, and then builds the correct syntax automatically. Modifying the syntax also updates the GUI, so users can switch back and forth between them. Users can order the attributes and methods in any way they wish. Default orderings of alphabetical, by access and by kind are provided.

Each method created in the class editor corresponds to a method tab under the corresponding class tab. The method editor begins with a blank workspace with a start and end symbol. The user can then add symbols corresponding to loops, selections, method calls, returns, assignments, inputs and outputs by selecting from the palette in the upper left corner and then inserting at an appropriate point in the program (see Figure 3). For convenience, a list of the attributes of the current class is displayed on the left-hand side of the method editor.

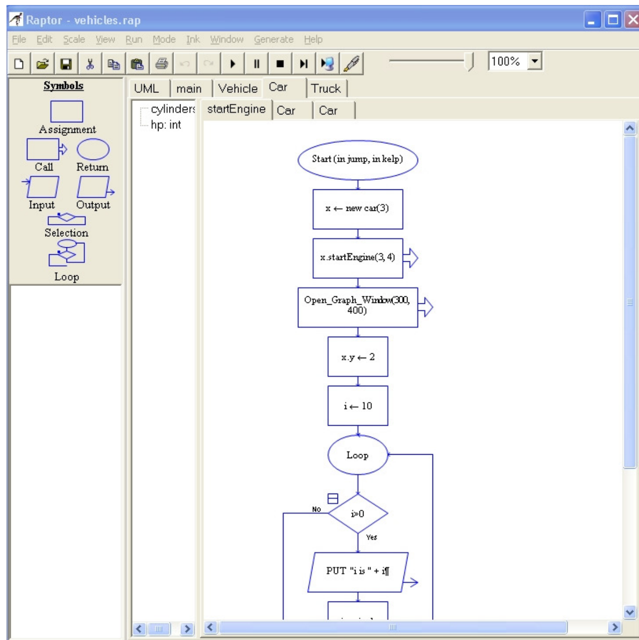

Figure 3: RAPTOR Method Editor.

RAPTOR methods are forced to be structured. Selections and loops must be properly nested, and each loop has a single exit point. The loop structure is modeled after the Java while loop. It is a pre-test loop, where the loop body is executed while the condition is true. The selection structure is modeled after the Java if-then-else. The left-hand side of the selection is the "then" branch, and the right-hand side the "else".

The syntax used within a symbol is designed to be flexible. Elements have been borrowed from both C and Pascal-style languages. For example, either "**" or "^" may be used as an exponentiation operation, and "&&" or "and" may be used as a Boolean "and" operator. RAPTOR enforces syntax checking on each symbol as it is edited. Therefore, it is impossible to create a syntactically invalid program. If the user enters "x+" as the right hand side of an assignment, they will get an error message and be required to fix the arithmetic expression before leaving the assignment box.

Commenting is done by right-clicking on a symbol and selecting "comment". The comment appears as a "talking bubble" next to the symbol. The comments can be clicked and dragged to improve the aesthetic of the program.

RAPTOR has over 80 built-in functions and procedures which allow the student to generate random numbers, perform trigonometric computations, draw graphics (including circles, boxes, lines, etc.), and interface with pointing devices. As seen in Figure 4, RAPTOR will automatically suggest completions to procedure names. Additionally, RAPTOR automatically suggests completion for user-created attributes and methods.

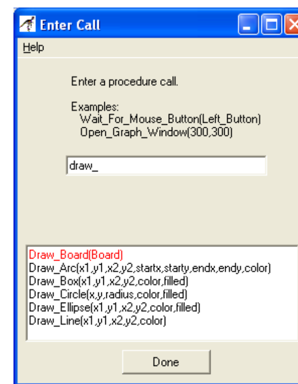

Figure 4: Entering a procedure call.

During execution, the student can select to single step through the program, or run continuously. The speed of execution is adjustable by moving the slider shown at the top of Figure 3. At each step, the currently executing symbol is shown in green. Additionally, the state of all of the variables is shown in a watch window at the bottom left corner of the screen. This window shows the entire call stack, and the heap. This allows instructors to easily demonstrate recursion, and also the difference between stack and heap variables.

To facilitate the transition to Java, we provide a Java code generator. This translates the UML diagram and all of the methods. This allows students to use RAPTOR to do design work and even some implementation, and then finish the code in a Java development environment. There is currently no facility for "round-tripping" (i.e. making modifications to the Java and then having those imported back into the RAPTOR design).

4. FUTURE WORK

In the upcoming semesters, we plan to further experiment with using RAPTOR to teach object-oriented programming by refining and expanding the programming assignments that we give to our students. In addition, we will continue to modify and improve the RAPTOR environment with richer sets of available functions and procedures, enhanced Help facilities, and other ideas to be gleaned from user feedback.

We would like to add the ability to import classes from other files, and provide pre-compiled classes (in the form of DLLs) that students can use in their programs. We would also like to add code generators for other languages (e.g. C++, C# and Visual Basic).

5. CONCLUSIONS

RAPTOR provides a simple environment for students to experiment with object-oriented programming. Instructors can use this to give students a visualization of object-orientation, recursion and heap vs. stack memory allocation. RAPTOR is the first free, open-source tool that fully supports introducing object-oriented programming, including the features of polymorphism and inheritance.

Prior results have indicated that students prefer visual representations and are more successful learning programming concepts when they are introduced using an iconic or flowchart form. RAPTOR allows us to leverage this success in introducing students to Java programming and object-orientation.

We have provided a web site where other universities can download RAPTOR. It is located at <URL omitted for blind review>.

6. REFERENCES

[1] *Computing Curricula 2001: Computer Science,* December 2001. Online [July 17, 2008]. Available at http://www.acm.org/education/curricula-recommendations.

[2] Cardellini, L. An Interview with Richard M. Felder. Journal of Science Education 3(2), (2002), 62-65.

[3] Fowler, L., Allen, M., Armarego, J., and Mackenzie, J. Learning styles and CASE tools in Software Engineering. In A. Herrmann and M.M. Kulski (eds), Flexible Futures in Tertiary Teaching. Proceedings of the 9th Annual Teaching Learning Forum, February 2000. http://ccea.curtin.edu.au/tlf/tlf2000/fowler.html

[4] Thomas, L., Ratcliffe, M., Woodbury, J. and Jarman, E. Learning Styles and Performance in the Introductory Programming Sequence. Proceedings of the 33rd SIGCSE Symposium (March 2002), 33-42.

[5] Scanlan, D. A. 1989. Structured Flowcharts Outperform Pseudocode: An Experimental Comparison. *IEEE Softw*are. 6, 5 (Sep. 1989), 28-36

[6] 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* (St. Louis, Missouri, USA, February 23 - 27, 2005). SIGCSE '05. ACM, New York, NY, 176-180.

[7] Cilliers, C., Calitz, A., and Greyling, J. 2005. The effect of integrating an Iconic programming notation into CS1. In *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Caparica, Portugal, June 27 - 29, 2005). ITiCSE '05. ACM, New York, NY, 108-112.

[8] Calloni, B. A., Bagert, D. J., and Haiduk, H. P. 1997. Iconic programming proves effective for teaching the first year programming sequence. In *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education* (San Jose, California, United States, February 27 - March 01, 1997). J. E. Miller, Ed. SIGCSE '97. ACM, New York, NY, 262-266.

[9] vanDijk, J. AdaGraph. Online [July 31, 2008]. Available at

http://users.ncrvnet.nl/gmvdijk/adagraph.html.

[10] Watts, T. 2004. The SFC editor: a graphical tool for algorithm development. *J. Comput. Small Coll.* 20, 2 (Dec. 2004), 73-85.

[11] *Visual Logic*. Online [July 31, 2008]. Available at: http://www.visuallogic.org.

[12] Ziegler, U., and Crews, T. An Integrated Program Development Tool for Teaching and Learning How to Program. Proceedings of the 30th SIGCSE Symposium (March 1999), 276-280.

[13] *Alice.* Online [July 31, 2008]. Available at: http://www.alice.org.

[14] Baldwin, R. Alice Programming Tutorial. Online [July 31, 2008]. Available at: http://www.dickbaldwin.com/alice/Alice0150.htm.

[15] 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* (Caparica, Portugal, June 27 - 29, 2005). ITiCSE '05. ACM, New York, NY, 104-107.

[16] *NClass.* Online [July 31, 2008]. Available at: http://nclass.sourceforge.net.