TEACHING COMPUTER PROGRAMMING WITH PROGRAM ANIMATION

Theodore S. Norvell and Michael P. Bruce-Lockhart
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University
St. John's, Newfoundland and Labrador, Canada A1B 3X5
theo@engr.mun.ca *and* mpbl@engr.mun.ca

Abstract

The Teaching Machine is a software system for animation of computer programs. It allows the instructor in the classroom or the student on their own to single step through computer programs written in C++ or Java while observing the effect of each step on the state of a virtual machine. The state of the machine is represented in a number of ways including a presentation of the current state of evaluation of an expression, the state of memory in terms of bits or symbolic values, a box and arrow view of pointer based data structures.

We report on recent developments in the Teaching Machine and on experience in the classroom.

Keywords: *Program Animation, Teaching Machine, Computer Education.*

1. INTRODUCTION

Students in early programming courses need to learn a conceptual model of program execution. That is they need a mental model of program state and a model of how that machine state is affected by the various statements, operations, and declarations within a computer program.

Some students get this conceptual model quickly and accurately from experience with programs in homework assignments or from reading the text and listening to the instructor. However, most instructors know that not all first-time students develop useful conceptual models easily or even at all.

We speculated that by showing a graphical representation of the program state and showing students how this evolved over time, under control of the execution of the associated program, the students would more easily form the appropriate mental models, The model can then be applied to learning new concepts: the mental model of execution, built while learning about if statements, can be applied when learning about while statements; the mental model of data structuring, built while learning about arrays, can be applied when learning about objects. These mental models could then apply to operational reasoning about their own programs as well as the examples that they have seen.

To test this hypothesis—and in hope of creating something useful for ourselves and for others—we developed a program animation system: the Teaching Machine.

An earlier paper [1], introduced the Teaching Machine; in the present paper we focus on the experience with the Teaching Machine in the classroom, on its design, and on recent improvements in language coverage.

Proceedings of the 2004 Canadian Conference on Computer and Software Engineering Education,

2. OUTLINE

The remainder of the paper is organized as follows. Section 3 reviews program animation with the Teaching Machine. Section 4 discusses the integration of the Teaching Machine with pedagogical web-sites. Section 5 recounts recent experience with the class-room use. Section 6 explains the architecture and design of the Teaching Machine, while Section 7 explains the extent to which it can handle the C++ and Java languages.

3. PROGRAM ANIMATION WITH THE TEACHING MACHINE

By program animation we mean that we create a graphical representation of the state of the computer. The execution of the program creates an animation in the sense of a series of snap-shots of the machine/program state.

Fig. 1 shows a snap shot of the Teaching Machine as it would typically be seen by a student in their first course in computer programming. We will explain the various "subwindows" shown in Fig. 1.

3.1. Source Code

To the left is the source code of a program written in C++. The line currently being executed is high-lighted.

3.2. Expression Evaluation

In the upper right, in the subwindow titled "Expression Engine," we see an expression which is partially evaluated. This partially evaluated expression is visually coded: parts that have been evaluated to values are shown in red; parts that have been evaluated to references are shown in blue, the next operation to be evaluated and its operands are underlined.

3.3. Symbol Table

Below the expression engine is a representation of the currently relevant part of the symbol table, showing the correspondence between variable names and addresses. When in the course of expression evaluation a variable needs to be converted to a reference, the corresponding line in the symbol table is high-lighted.

Of course in a compiled language like C++, there is no run-time representation of the symbol table, except perhaps for debugging purposes. However we find that including this aspect of the compiler state in the run-time model is useful for beginning students. As students mature in their understanding, we expect that this blending of compile-time and run-time concepts will be understood as such.

3.4. Memory

In the lower right is a subwindow labelled "Memory". This shows the mapping of addresses to variable values. Memory is shown at a level of abstraction suitable for students: `int`s are shown in decimal, `char`s as glyphs. Structured variables such as arrays, `struct`s, and class members can be expanded or contracted to show or not show their components.

Pointers are represented as decimal numbers and references as the name of the variable they represent. Memory can also be shown in terms of bits, which can be useful for showing that pointers and references really share the same representation. Sometimes the best way to illustrate an abstraction is to break it.

When a location in memory is about to be accessed, either for reading or writing, the item in the "Memory" subwindow is highlighted.

As shown in the figure, "Memory" actually corresponds to the stack (local memory). In more advanced courses, the stack, heap and static memory are all tracked separately.

3.5. Other Displays of State

In the standard views of memory, pointers are shown as addresses (in decimal) and references are represented by the name of the variable being referenced. An alternative view of memory is shown in the Linked View window, illustrated in Fig. 2 and Fig. 3. The linked view shows stacked data on the left and heap data to the right. Pointers and references are shown as arrows pointing to the box representing the data item they point to.

3.6. Control of Execution

The program can be stepped at a variety of levels of granularity. Various buttons and menu items allow control of the Teaching Machines execution. One can

- "Go Forward" Step to the next expression evaluation step. An expression evaluation step is the lookup of a variable address in the symbol table, the lookup or storage of a value in memory, the evaluation of a single operator, or conversion. Each step is illustrated in the Expression Engine.

- "Step into" Step to the next expression, which might be within a called subroutine.
- "Step over" Step to the next expression, skipping over any subroutine calls.
- "Go to cursor" Step to a user selected line.
- "Go back" Undo the last stepping command. Undo is limited only to the current program execution.

The "Go back" command can be particularly useful for students who want to know "What just happened?". They can review a previous state or replay a stage of execution again, perhaps at a finer level of granularity.

3.7. Program animation and debugging

From a technical standpoint, the Teaching Machine bears resemblance to a debugger. However the intended audience and purpose is quite different. A debugger is intended for a professional software engineer who already has a good understanding of programming. The Teaching Machine is intended to help learners to build effective mental models of programming. The linked-view gives an example of this. In a debugging context, the linked-view would be hopelessly space consuming and would require support for navigation through structures too big to fit on the screen. In the Teaching Machine, the linked view is only intended for small examples and works quite well.

4. INTEGRATION WITH THE WORLD WIDE WEB

Inasmuch as the Teaching Machine is written in Java and can therefore be run as an applet it seemed natural to embed it directly into teaching web pages. In the first instance, these were conceived of as tutorial pages to help students who either were taking a traditional course and needed extra help, or who needed to brush up on their understanding. The process of creating those first pages quickly convinced us that more than just the Teaching Machine was needed.

A major shortcoming of HTML was the difficulty of displaying code examples well on a web page. What we felt was needed was a number of authoring aids aimed at instructors of programming. Thus was born a second tool.

WebWriter++ is a small authoring system written in Javascript that allows instructors, working with any HTML editor, to create web pages for programming courses easily. Its most important feature is that it allows example C++ and Java source files to be dynamically retrieved across the net and displayed on a web page. The examples can be prepared separately, compiled and debugged, as well as being re-edited later on. The examples are lexed and displayed as they would be in a program editor, with keywords, comments, and constants all marked. By simply changing the site stylesheet, the instructor can match their appearance to whatever program editor happens to be in use in the course.[1]

Further, using a simple markup system embedded in comments in the code, the instructor can select only a portion of the code for display. For example, we commonly show a single function, discuss it, then show its calling context and discuss that. If the person viewing the page wants to run the example, WebWriter++ provides buttons for running the example in the Teaching Machine, as well as possibly viewing a video of the example being run in the Teaching Machine, if one is available. Since exactly the same source file is displayed on the web page as is loaded in the Teaching Machine, there is no problem with keeping examples in sync. These buttons can be seen on the WebWriter++ page in Fig. 4.

The authoring system includes a number of other features—indeed, it is used as a test bed to try a variety of techniques for effectively teaching via the web. For example one can roll the mouse pointer over a variable and see its scope illuminate in the code.

5. EXPERIENCE WITH CLASSROOM USE

In the fall of 2002 we started to integrate the Teaching Machine directly into the class notes for our Advanced Programming course—the second programming course of Memorial University's ECE programme. For years the instructor (Bruce-Lockhart) lectured from transparencies, with the Teaching Machine being turned to from time to time. All notes were migrated to HTML using WebWriter++ and examples were integrated directly into them. The notes were then projected to the classroom and printed versions made available to the students. In 2002, the conversion process was carried out while the course was being taught, so that notes could not be made available all at once. Moreover, the projector was of sufficiently low luminance that all the lights had to be turned out in the lecture hall. Nevertheless, students were guardedly positive.

[1] The Teaching Machine also allows appearance to be customized.

In fall 2003 the students were able to get a bound copy of the notes on day one as well as a CD of the website. And the course was moved to a new room with a new projector that enabled normal ambient lighting.

An unlooked for side effect was that the instructor, no longer in the dark and not chained to a board or an overhead, was able to stroll around the room, laser pointer in hand, and engage with students in a much more direct fashion than had been possible before. The course was far more fun to teach than it had been in the past. The students liked it as well.

The Chair for Electrical & Computer Engineering, who conducted exit interviews with our third term students, somewhat ruefully confided to the instructor that they had said the course was "perfect" and that even the ones who didn't like programming "could not conceive of the subject being taught any better." While such praise is gratifying, what had changed was not the instructor but the first full integration of the Teaching Machine into the course.

Formal teaching reviews have just been received. The instructor's approval rating has gone up significantly (from 4.15 before the integration to 4.6). In addition, optional comments were almost uniformly positive. All comments received about the Teaching Machine and/or the notes were positive except two (which made complaints about content not presentation). Several students said the Teaching Machine should be used in our earlier course, and in fact we are in the process of converting that course over, even as this is written.

6. SOFTWARE ARCHITECTURE AND DESIGN

Figure 5 shows the software architecture of the Teaching Machine. It can be viewed as consisting of an executive layer and three subsystems

- The Executive layer handles the main frame and menus of the Teaching Machine. It also mediates between the Display System and the Virtual Machine.
- The Language Stack deals with all language dependant aspects. Parsing is done with a recursive descent parser generated by JavaCC [2]. The Parser and Analyser form a compiler which produces a graph-structured representation of the program. The nodes of these graphs are instances of classes from the AST[2] layer. The AST classes not only represent the structure of programs, they also contain the behaviour of the languages various operations and data-types. At run-time data is represented by "Datum objects" belonging to classes drawn from the Datum layer.
- The Virtual Machine consists of an Evaluator and a Virtual Machine State. The Virtual Machine State represents the state of the virtual machine at run time. It contains all the memory and various collections of Datums and various stacks, the most crucial of which is a stack of Evaluations.

  Each Evaluation represents the partial execution of one expression or function. Each evaluation contains the graph representing the expression or function body, optionally a selected node, and a partial labelling of the nodes in the graph with objects. In the case of expressions, nodes are labelled with references and data values; as the expression is evaluated the labelling propagates from the leaves to the root. At run-time, the Teaching Machine advances by selecting a node in the current graph, if none is selected, and otherwise asking the currently selected node to execute one execution step.

  The Evaluator's main job is to interpret the various user commands so that the state is driven forward just the right amount.
- The Display System implements most of the Teaching Machine's look. The Display Engine converts the virtual machine state to images on the screen, while the Subwindow layer provides a layer on top of Java's AWT packages to provide windows within windows.

The original language subsystem supported a very simple subset of C++ [1]. The front-end contained only a parser, so all analysis was postponed to run-time. This arrangement was fine for the simple subset supported, but was too limited for much of C++. For example compiler generated methods could not be supported in any

[2] AST abbreviates Abstract Syntax Tree. However this terminology reflects an early language stack in which there was no analyser and all analysis was done at run time. For example the "<<" operator of C++ was always represented by the same AST class, regardless of whether it represented a shift or an output command. In more recent language stacks, extensive compile-time analysis is done and the AST code is really a graph-structured intermediate code.

reasonable way. Furthermore, as analysis was being done at run time, the AST nodes (which encapsulate run-time behaviour) could not be language independent.

We therefore embarked on a complete reimplementation of the language subsystem with the following goals: Better C++ language coverage, close to complete Java coverage, proper compile-time analysis, lower-level and simpler AST code, and extensive sharing of code between the Java and C++ language stacks. Each layer of the language stack is split into three Java packages: one for the C++ implementation, one for the Java implementation, and one for classes used in both implementations. At the AST layer the majority of the code is common.

Throughout the design, design patterns [3] are used extensively. Two examples are given here. Compilation makes extensive use of the Command pattern; as each operator is encountered, a code generation command is looked up based on the operator and the operand types. At run time, the Abstract Factory pattern is used to recursively generate datum objects from the trees representing types; type trees are factories for datum trees.

7. LANGUAGE COVERAGE

As can be seen from Table 1, a reasonable "teaching subset" of C++ is currently covered however, there are some major limitations, so one can not expect every teaching example to run, nor for students in more advanced classes to run all their own code.

Our Java implementation is nearing the point of usability in the class room. Our aim for Java is to cover all the language and to create a bridge to the Java libraries so that user code can extend and use library classes.

8. CONCLUSIONS

Does the Teaching Machine help students to build effective mental models of program operation? We have no direct evidence, but we do have evidence that its use has helped to ease what is often a difficult subject for many students.

Animations have been shown to be helpful in self-tutorial situations [4,5], under the right circumstances. We have used the Teaching Machine mainly as a tool in the class-room while making it available for students to use for self-study. Our experiences have been very positive, especially when the use of the Teaching Machine is seamlessly integrated into the course notes and lectures.

Table 1 C++ Language Coverage

Currently supported • All primitive types and operators • Functions definitions and function calls • Most statements • Variable declaration • Class declarations • Compiler generated members • #include directives • Some library
Anticipated • Exceptions • User defined conversions • Virtual functions • Namespaces • In-place function member definition • Calling destructors • Complete preprocessing • Multiple compilation units • More library
Not anticipated • Templates

Acknowledgements

Much of this research was carried out under a grant from the Office of Learning Technology, Department of Human Resources and Development.

We are greatly indebted to Derek Reilly for his work on the new Language Stack. Derek deserves a medal for Code Heroism.

We would also like to thank the Faculty of Engineering at Memorial University for supporting new teaching methods, and providing us a working environment.

References

[1] M. P. Bruce-Lockhart and T. S. Norvell, “Taking the hood off the computer: Program animation with the Teaching Machine,” Canadian Conference on Electrical and Computer Engineering, Halifax, N.S., 2000.

[2] S. Viswanadha and S. Sankar , “JavaCC”, https://javacc.dev.java.net/, 2003.

[3] E. Gamma, R. Helm, Ralf Johnson, John Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[4] R. Ben-Basset Levy, M. Ben-Ari, P. Uronen, “The Jeliot 2000 program animation system, *Computers & Education*, vol. 40, pp. 1—15, 2003.

[5] T. Ellis, “Animating to build higher cognitive understanding: A model for studying multimedia effectiveness in education,” *J. of Engineering Education*, vol 93, #1, pp. 59—64, 2004.

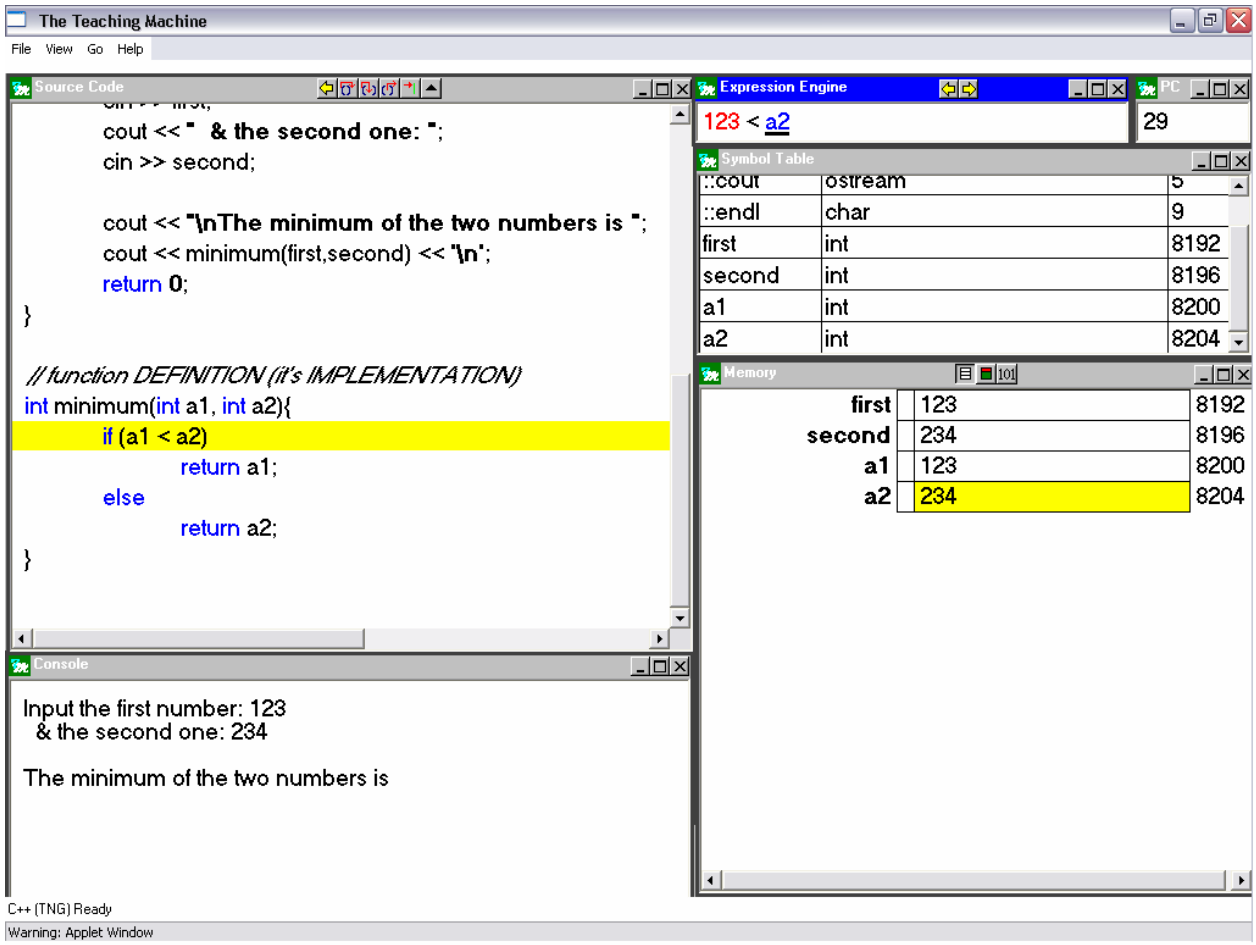

Figure 1 A First Course Example

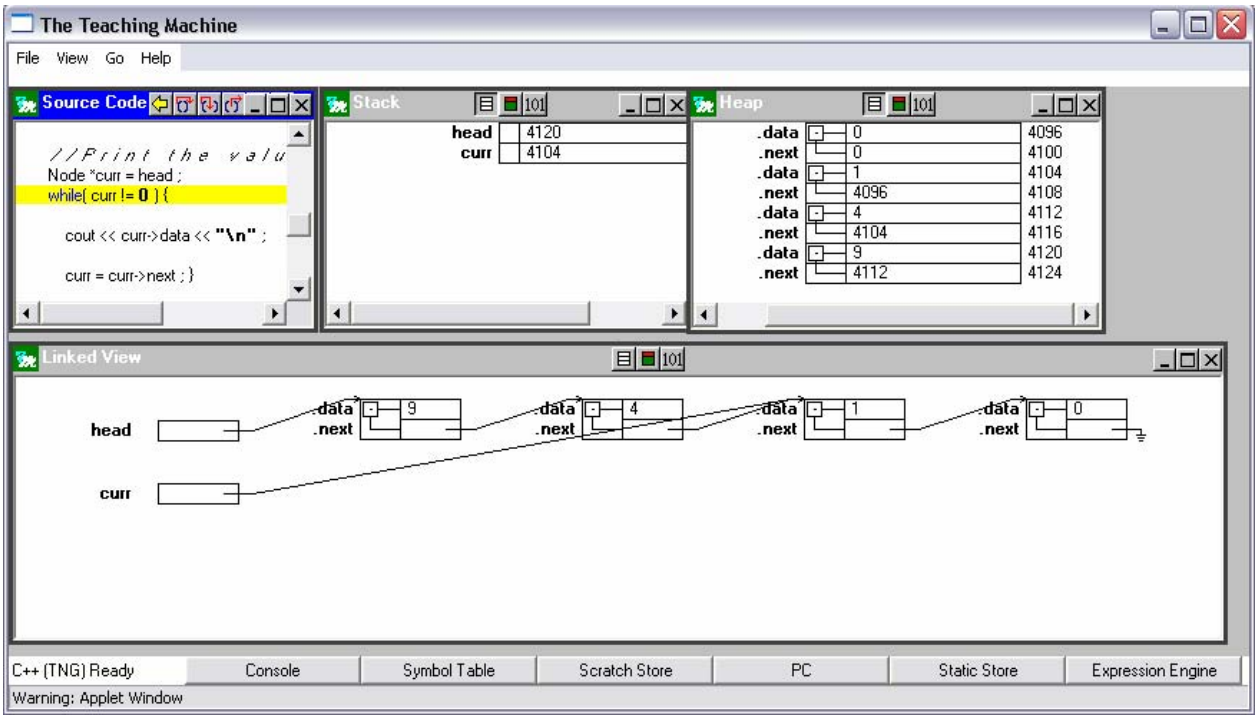

Figure 2 Linked View applied to objects on the heap

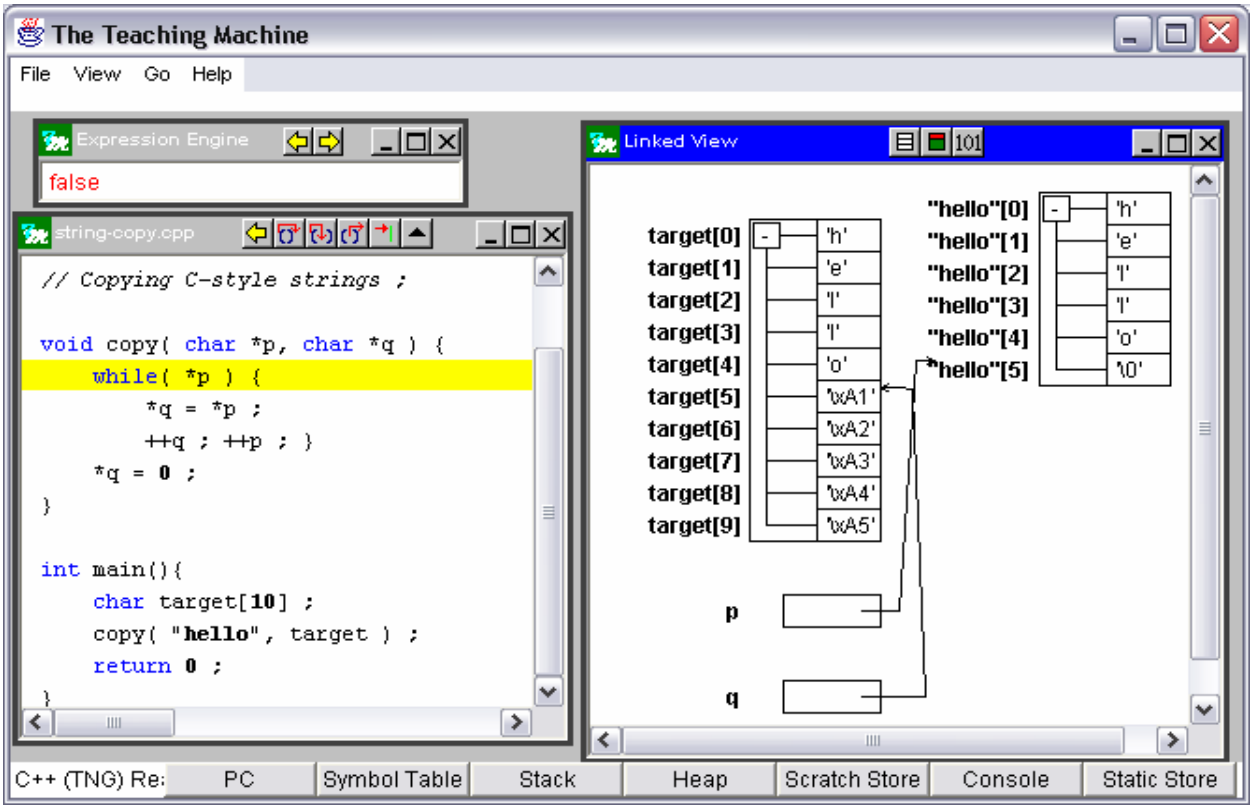

Figure 3. The linked view applied to arrays

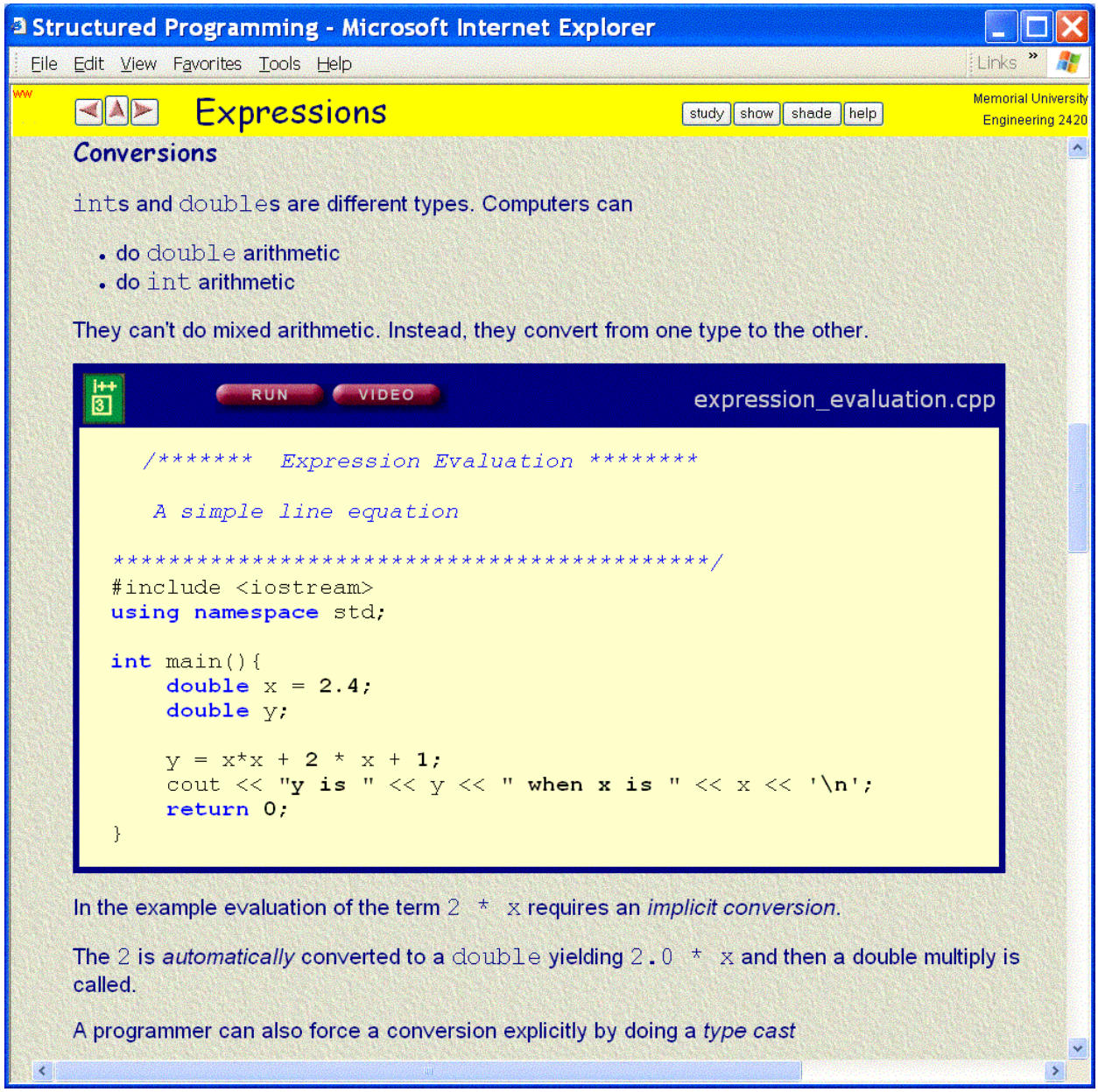

Figure 4 WebWriter++

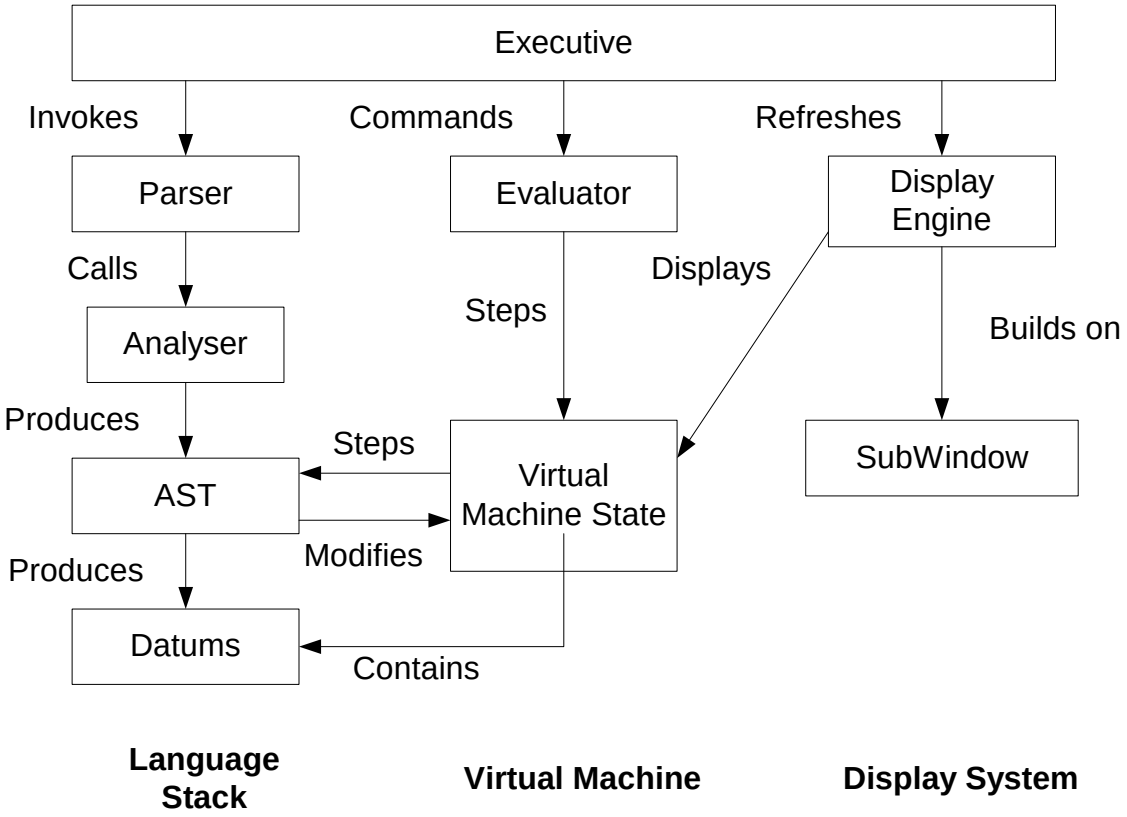

Figure 5 Main components and relationships