BlackBox: A New Object-Oriented Framework for CS1/CS2

J. Stanley Warford
Pepperdine University
warford@pepperdine.edu

1. Abstract

This paper describes the BlackBox framework, an object-oriented application development environment, and our experience with its use over the past several years in the CS1/CS2 course. This little-known framework features: (1) a graphical user interface that is simple enough for beginning students to program, (2) true cross-platform capability, (3) guaranteed memory-safe pointers with automatic garbage collection, (4) a new language, Component Pascal, that combines the best of Java and Pascal, (5) fast, native-code compilation, and (6) lowest possible cost (free to educational users). The paper concludes with a guide to further resources for those who wish to pursue this promising new framework in their curricula for the first year.

1.1 Keywords

BlackBox, Component Pascal, CS1, CS2, frameworks, programming languages, formal methods.

2. Introduction

In the Fall semester of 1996, we began an experiment with a new programming language and application development environment for our introductory computer science courses. The environment was called Oberon/F, and the language on which it was based was called Oberon/L. Both the language and the framework changed significantly the following year. The framework is now named BlackBox Component Builder and the language has evolved to Component Pascal. The pedigree of the language is:

Pascal → Modula → Oberon → Component Pascal

The power of the system lies in the simplicity of the programming language and the synergy that it has with the framework. The development environment was designed for professional programming, and its emphasis on components (sometimes referred to as "beyond objects") is for programmers with expertise much greater than what would be expected of students in an introductory course.

However, we have discovered that the combination of the simplicity of the language and the power of the framework combine to make an excellent vehicle to carry students through the first year of our curriculum. By the time this paper is published, we will be completing our third year of what has been an eminently successful experiment.

This paper begins by describing the philosophy of our curriculum and how the framework and language support that philosophy. The following sections describe those aspects of the language and the graphical user interface that are incorporated into the CS1/CS2 course. The paper concludes with a guide to further resources.

3. The CS1/CS2 Curriculum

Our department recently reviewed the computer science curriculum and identified four areas of the discipline that needed to be integrated more fully into the course of study:

- Human/computer interaction
- Formal methods
- Object-oriented programming
- Computer networks

The new curriculum is designed to introduce the first three of these areas during the freshman year. Formal methods is presented in a separate course that runs parallel with CS1/CS2. BlackBox is the tool in CS1/CS2 with which we teach human/computer interaction using the framework's graphical user interface, and Component Pascal is the vehicle for teaching object-oriented programming.

3.1 Philosophy of the Curriculum

Our curriculum is based three themes

- abstraction
- integration
- languages and paradigms

Two of these themes—abstraction, and languages and paradigms—are relevant to our use of BlackBox in the introductory course.

3.1.1 Abstraction

Abstraction is based on the concept of layers in which the details of one layer of abstraction are hidden from layers at a higher level. A computer scientist uses abstraction as a thinking tool to understand a system, to model a problem, and to master complexity. The ability to abstract cannot be acquired in a single course, but must be developed over several years. Consequently, all courses in the curriculum emphasize the abstraction process, not only as a framework to understand the discipline but also as a tool to solve problems.

3.1.2 Languages and paradigms

Because of the continued evolution of programming languages and paradigms we would do our students a disservice by emphasizing only one programming language or paradigm throughout the curriculum. Students should be multilingual and should experience multiple paradigms in their undergraduate careers. Our curriculum seeks to strike the proper balance between breadth and depth. Too much breadth will not equip students with the detailed skills necessary to solve realistic problems. Too much depth in one language or paradigm will give students a narrow vision that makes it difficult to consider multiple approaches to a problem.

The curriculum emphasizes in-depth proficiency the first two years and more breadth the last two years. The balance is achieved by choosing one programming language for both semesters of the first year and another language for both semesters of the second year. Courses in the third and fourth years introduce other programming paradigms based on different languages.

The language choice for the first year is driven primarily by pedagogical concerns. Pedagogical concerns are important during the first year, since this is when students begin to form algorithmic thinking patterns and develop problem-solving skills. The criteria are that the language should be simple to learn yet powerful enough to illustrate fundamental concepts of computing. The language for the second year is a mainstream language widely used in the industry (C++). Skill in a practical language is necessary for students to be well equipped for their post graduate careers. The languages for the third and fourth years (primarily Java, but also Lisp and Prolog) are chosen for the variety of programming paradigms on which they are based.

3.2 CS1/CS2 Topics

The selection of BlackBox as the development environment and Component Pascal as the programming language is motivated by the philosophy expressed in the previous section. Unlike Java or Smalltalk, Component Pascal is by design a hybrid language that incorporates both the procedural and the object-oriented paradigms. In this respect, it is more akin to Ada and C++, which are also mixed-paradigm languages. However, its type safety and memory protection with automatic garbage collection make much of its semantics closer to Java than to either Ada or C++.

Our CS1/CS2 course follows a sequence of topics whose goal is to guide the student from the imperative paradigm to the object-oriented paradigm. The framework permits students to begin with the procedural approach to programming, continue through successively higher levels of abstraction, and culminate with the object-oriented paradigm. It therefore satisfies our goal of students learning multiple paradigms in the curriculum.

Some will find our curriculum philosophy and its implementation beginning with the procedural paradigm and advancing through successively higher levels of abstraction to the object-oriented paradigm to be controversial at best and ill-conceived at worst. It is beyond the scope of this paper to engage in the religious debate between those who believe that a pure object-oriented language should be taught at the outset and those who favor a journey from low to high levels of abstraction.

Our choice of the latter approach is based on the belief that to truly understand abstraction, students must experience it at several levels. BlackBox provides that experience by its elegant use of interfaces in the framework, which permit students to use objects from the outset. Furthermore, because the Component Pascal language is not pure object-oriented, it permits this approach of using objects within a procedural paradigm before implementing objects and programming with the full power of polymorphism at the end of their one-year journey.

1. The BlackBox Framework
2. Languages and Grammars
3. Modules and Interfaces
4. Variables
5. Dialog Boxes
6. Abstract Stacks and Lists
7. Selection
8. Nested Selection
9. Objects and the MVC Paradigm
10. Loops
11. Nested Loops
12. Proper Procedures
13. Function Procedures
14. Random Numbers
15. One-Dimensional Arrays
16. Stack and List Implementations
17. Iterative Searching and Sorting
18. Two-Dimensional Arrays
19. Recursion
20. Recursive Searching and Sorting
21. Linked Lists
22. Binary Trees
23. Objects and Methods
24. The State Design Pattern

Figure 1: CS1/CS2 Topics

Figure 1 shows the sequence of topics in the first-year course. Programming to an interface is mandatory from the beginning because of the structure of the framework. By the third week of the first course, students program with dialog boxes. Before loops are introduced, students learn how to use the model/view/controller (MVC) paradigm on which the BlackBox windowing system is based. From this point on, the sequence of topics is similar to the sequence in a course based on a traditional command line interface, except that student are applying the topics in a graphical user interface environment.

4. Component Pascal

Component Pascal is typical of the languages designed by Niklaus Wirth—small, simple, elegant, yet powerful. Here are some characteristics of the language:

Modules—The module is the basic unit of compilation. A module can contain one or more classes and can export types, constants, variables, procedures, classes and objects. Any item not exported corresponds to a protected field in C++/Java, except that the protection extends over the entire module, not just a class. There are two modes of export, read-only and read/write. An item that is exported read-only can be accessed but its value cannot be changed by the importing module. This feature eliminates the need of methods whose sole purpose is to return the value of a state variable of a class.

Interfaces—Interfaces are central to the language. Unlike header files in a C++ environment, the compiler, rather than the programmer, creates the interface of a module. Hence, the programmer need not worry about textual consistency between her module and its interface. The framework automatically keeps track of consistency between compiled and recompiled modules.

Memory protection—The C++ language provides explicit pointers but cannot insure against memory leaks. The Java language does not provide explicit pointers, but guarantees memory protection with its automatic garbage collection. Component Pascal provides both explicit pointers as well as automatic garbage collection. Only type-safe operations on pointers are allowed (i.e. no pointer arithmetic), and all pointers are initialized automatically to NIL. There is no memory delete or dispose operation. Component Pascal provides run-time checks on array bounds.

Types—The primitive numeric types are identical to those of Java. Integers are defined as fixed-range, platform-independent 32-bit quantities. Reals are IEEE double-precision. Characters are Unicode. Because Component Pascal is not a pure object-oriented language, it retains arrays and records as primitive types. Strings are null-terminated arrays of characters, not objects, and the + operator provides the concatenation operation. The $ modifier provides efficient assignment of strings. If a designates an array of character type, then a$ denotes the null terminated string contained in a. The equivalent of C's library call to copy strings is unnecessary.

Parameter passing mechanisms—There are four parameter passing mechanisms: call by value (default), call by constant reference (IN), call by result (OUT) and call by reference (VAR).

Object-oriented—Component Pascal is fully object-oriented with polymorphism and single inheritance. The syntax for methods differs from most object-oriented languages. There is no need for the concept of "self" or "this" in a method because the object of the method is passed as an explicit parameter. Templates are not yet part of the language, but may be in the future [2].

Dynamic linker/loader—The compiler generates native code. There is no virtual machine or byte code intermediate language. Modules are loaded on demand within the framework. The module architecture eliminates the need for the static members of Java. Because the module is the unit of compilation, and a module can contain more than a single class, it can contain the single persistent state variable or object shared by all the other objects of a class.

Debugging—Each method or procedure in BlackBox is documented with pre- and postconditions. When a precondition is violated or a run-time error occurs, a trap window is generated that provides a snapshot of the run-time stack at the time of the fault. The trap window contains the current value of all the variables and hypertext links to the offending source code. Component Pascal supports this style of documentation with the ASSERT statement.

These features of the language further the goals of our curriculum. The central roles of modules and interfaces support the concept of abstraction as one of its themes. Automatic garbage collection and native array bounds checking minimize errors beginning programmers can make, and provide helpful support when run-time errors occur.

Because the language in the second year is C++, Component Pascal provides a convenient transition to that language. As in C++ and Ada, there is more than one parameter passing mechanism, pointers are explicit, and the language embraces without apology a mixed procedural/ object paradigm.

The ASSERT statement in the language furthers the goal of an early exposure of formal methods. Students learn to program to a specification with pre- and postconditions and learn how to use assertions to establish preconditions for their own procedures and methods. Connection is made to the Hoare Triple of formal methods in the parallel course.

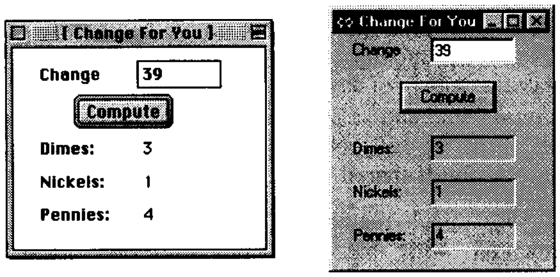
5. The Graphical User Interface

Using an object-oriented framework in the first year caused us to reassess the entire issue of input/output. We reconsidered what is important for students to learn about the human/computer interface. As it turned out, there was a natural one-to-one replacement of topics. In place of interactive I/O, where the program prompts the user for input with a command line interface, is dialog box I/O. In place of file I/O is window I/O using the MVC paradigm.

5.1 Dialog Box I/O

One of the most gratifying experiences with BlackBox in the introductory sequence is the ease with which students can program a modern GUI. Figure 2 gives an idea of how easy it is to program a dialog box. The same program compiles without change on a Windows or Mac platform. Indeed, students are free to use whichever platform they like, and either platform can be used to test students' programs and dialog boxes. Dialog boxes retain the host platform look and feel, even if originally developed on a different platform.

Dialog boxes are an example of BlackBox component


```
[ Change For You ]
Change   39
Compute
Dimes:    3
Nickels:  1
Pennies:  4
```

```
Change For You
Change   39
Compute
Dimes:    3
Nickels:  1
Pennies:  4
```

```
MODULE Sigcse99Fig2;
    IMPORT Dialog;

    VAR
        d*: RECORD
            change*: INTEGER;
            dimes-, nickels-, pennies-: INTEGER
        END;

    PROCEDURE MakeChange*;
        VAR
            cents: INTEGER;
    BEGIN
        cents := d.change;
        d.dimes := cents DIV 10;
        cents := cents MOD 10;
        d.nickels := cents DIV 5;
        d.pennies := cents MOD 5;
        Dialog.Update(d)
    END MakeChange;

BEGIN
    d.change := 0;
    d.dimes := 0; d.nickels := 0; d.pennies := 0
END Sigcse99Fig2.
```

Figure 2: A Component Pascal program and its dialog box.

containers. Students design them with graphic layout tools and link their controls to exported items of a module. In Figure 2, d.change is exported read/write (with the *), and linked to the input field of the dialog box. d.dimes, d.nickels, and d.pennies are exported read-only (with the -) and linked to the output fields of the dialog box. Procedure MakeChange is linked to the button labeled Compute. We discovered that the process of programming with dialog boxes is so simple in Component Pascal that programs are frequently shorter than equivalent programs in other languages using a command line prompt.

Figure 2 shows that BlackBox is a true framework, not just a library with a collection of classes and objects. The event loop is completely hidden. There is no main program. This central characteristic of a true framework is frequently called "The Hollywood Principle", that is, Don't call us, we'll call you. Note that the framework calls the procedure in response to the user clicking a button on the dialog box. The programmer does not write a main program, which in turn calls the procedure.

5.2 Window I/O with the MVC Paradigm

In BlackBox, files take a secondary role to the GUI and are available at a lower level of abstraction than are windows. Although it is possible to perform file I/O in BlackBox we decided that students' time would be better spent learning the MVC paradigm on which the windowing system is based. The MVC technique, pioneered at the Xerox Palo Alto Research Center in conjunction with the Smalltalk language, has proven its worth in many systems and is referred to extensively in [1] and [3].

The primary design concept in the MVC paradigm is the dissection of a data object into three parts—its model, which contains the data, its view, which presents the data on a display device, and its controller, which controls the interaction between the user and the view.

A full understanding of the MVC paradigm is obviously beyond the reach of beginning programmers. However, the basic concept is not that difficult to grasp. Furthermore, programs that perform window I/O are easy to write and understand in Component Pascal with a minimum of handwaving. Figure 3 shows a program presented early in CS1.

```
MODULE Sigcse99Fig3;
    IMPORT TextModels, TextControllers, PboxMappers,
        PboxStrings, Out;

    PROCEDURE ComputeTotal*;
        VAR
            md: TextModels.Model;
            cn: TextControllers.Controller;
            sc: PboxMappers.Scanner;
            balance: REAL;
            sum: REAL;
            sumString: ARRAY 16 OF CHAR;
    BEGIN
        cn := TextControllers.Focus();
        IF cn # NIL THEN
            md := cn.text;
            sc.ConnectTo(md);
            sum := 0.0;
            sc.ScanReal(balance);
            WHILE ~sc.eot DO
                sum := sum + balance;
                sc.ScanReal(balance)
            END;
            PboxStrings.RealToString(sum, 1, 2,
                sumString);
            Out.String("Total is $");
            Out.String(sumString); Out.Ln
        END
    END ComputeTotal;

END Sigcse99Fig3.
```

Figure 3: Using the MVC paradigm for input from a window.

The program assumes that a window is open and contains a sequence of real numbers that represent dollar amounts. No sentinel at the end is required, nor is an initial count of how many real numbers are in the list. The procedure is activated by the user selecting a menu option.

In the first statement of the procedure, text controller cn attempts to connect to the focus window. If the focus window contains text, cn will not be NIL, and the body of the IF statement executes. cn is an object that contains (class composition) a reference (exported read-only) to its model. In the first statement in the body of the IF, md gets the reference to the model.

Object sc is a scanner, an iterator that traverses the text model. The next statement connects the scanner to the model. Its position is now before the first character of text. Performing a scan when the iterator is at the end of text does not trigger a trap. It simply sets sc.eot, a boolean field of the scanner exported read-only, to TRUE. Hence, the program terminates correctly even if the focus window is empty or contains only white space.

The first year we used the framework we attempted to use the scanner supplied by the BlackBox framework. It is powerful for experienced programmers to use, but proved too difficult to understand for beginners. Scanner sc in Figure 3 is an extension (using the decorator pattern of [1]) of the BlackBox scanner, and is designed for the introductory course. The scanner can scan integers, reals, characters, strings, one-dimensional arrays of integers and reals, and two-dimensional arrays of integers and reals.

Space limitations preclude showing an example of a program that creates a new text model, attaches a view to it, and opens the view in a new window. Iterators for modifying a text model are called formatters, and the methods are similar in concept to formatted writes in other languages. For example, if fm is a text formatter the statement

```
fm.WriteReal(x, 10, 2)
```

writes to the text model the value of real variable x with a field width of 10 and 2 places past the decimal point.

6. Conclusion

There is a real synergy between the BlackBox Component Builder framework and the Component Pascal language. The language is not a toy designed only for pedagogical use, as evidenced by the fact that the entire framework including the compiler is written in Component Pascal.

Nevertheless, the framework has proved to be an ideal vehicle to carry our students through the CS1/CS2 sequence. The language is simple and small with clean syntax, yet has the power of full object-orientation with polymorphism. Students are excited to learn how to program a modern GUI, instead of being bored with the old command line interface. The language and the framework combine the best of Pascal and Java and deserve serious consideration as tools to further our educational goals at the CS1/CS2 level.

6.1 Resources

One of the best features of BlackBox is that the complete development system is available from Oberon microsystems at

http://www.oberon.ch/

and is free for educational use. The on-line documentation contains the defining language report and a sequence of tutorials (although geared to the experienced programmer, not the typical CS1/CS2 student). Applications written in BlackBox with the educational version must be run from within the framework. The company derives its revenue by selling a developer's version that enables the construction of a stand-alone executable application. The educational version has the full programming capability of the developer version.

The largest obstacle we had to overcome was the lack of a textbook appropriate for the CS1/CS2 sequence. A manuscript [4] is in development with chapters available at

ftp://ftp.pepperdine.edu/pub/compsci/prog-bbox/

in PDF format. Figure 1 is essentially a list of chapter titles of the manuscript.

7. References

[1] Gamma, E, Helm, R, Johnson, R, Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] Roe, P, Szyperski, C. Lightweight Parametric Polymorphism for Oberon. Fourth Joint Modular Languages Conference (JMLC'97), Linz, Austria, March 1997.

[3] Szyperski, C. *Component Software—Beyond Object-Oriented Programming*, Addison-Wesley and ACM Press, 1998.

[4] Warford, S. *Programming in BlackBox.* Prepublication, Pepperdine University, 1996.