

Object Oriented Spline Software

R. Bartels

Abstract. Object oriented programming provides software with a facility for imitating the mathematician's tool of theoretical abstraction. The commonality of a collection of related entities can be described in terms of a *base class*, which may be added to an object oriented language as a new data type. Each concrete entity of the collection can be described as a *derived class* by specifying whatever features distinguish it from the base, and the entity can be added as a further data type. Algorithms written for the base type can be applied to any of the derived types. This paper provides an example of these concepts applied to the design of a library of spline evaluation and refinement tools.

§1. Object Oriented Programming

The three fundamental concepts of object oriented programming are: *encapsulation*, *inheritance*, and *polymorphism*. Together they provide mechanisms for the programmer to extend the programming language by introducing new data types and operators. More importantly, they provide a means of writing code that is generic for collections of related data types.

Encapsulation allows one to define bundles of data and procedures, with associated operators, as new data types in the language. Variables with this type can be defined and used in programs as easily as the types provided originally. Inheritance allows one to define new data types in terms of those already defined, simply adding to the existing type the few additional items of data, procedures, and operators that distinguish the new type, and/or redefining data, procedures, and operators from the existing type. Polymorphism allows one to write code that can be executed generically on a collection of related data types.

Curves and Surfaces II

1

P. J. Laurent, A. Le Méhauté, and L. L. Schumaker (eds.), pp. 1–7.

Copyright © 1991 by AKPeters, Boston.

ISBN 0-12-XXXX.

All rights of reproduction in any form reserved.

In the next section we shall give examples using C++. The examples will be kept unrealistically simple, just containing enough components to illustrate the main points. For a general treatment of object oriented programming, see [3], and for the truth about C++, see [5].

§2. Examples

In C++ the definition of a type is a *class*. A simple example would be a type to represent squares, which could have the following declaration:

```
class Square
{
public:
    Square( void )
        { _left = 0.0; _bottom = 0.0; _width = 1.0; }
    void left( double x )
        { _left = x; }
    double left( void )
        { return( _left ); }
    ...
    double area( void )
        { return( _width*_width ); }
protected:
    double _left, _bottom, _width;
};
```

This class contains most of the conventional elements. It consists of eight procedures and three data items. In the interests of space, four of the procedures have been omitted in the location indicated by triple dots. The procedures left out are two versions each of `bottom` and `width`, whose definitions are similar to those of `left`. (The capacity to have multiple definitions for a procedure or operator, each distinguished by different argument types, is called *overloading* and is supported in C++.) Together the procedures and data are referred to as the *members* of the class. The data members `_left`, `_bottom`, and `_width` constitute the major aspect of the class implementation; *i.e.*, they reflect the decision to represent a square by its width and the coordinates of its lower left corner. Any outside user of the class is prevented from having direct access to these members by registering them as `protected`. Indirect access is available through the interface provided by the member procedures `left`, `bottom`, and `width`, each having two definitions, one for setting a value and one for reporting the value, and all are registered as `public`.

Indirect access is preferred, since it allows the designer of a class to change the class implementation without disturbing the use of the class; *e.g.*, to switch to a representation that uses the coordinates of the center of the square, if some future consideration should reveal that as being more efficient. The code for a member procedure such as `left` would be changed appropriately, of course, but the user of the class need never know that a change had

taken place. This capacity to separate *interface* cleanly from *implementation* constitutes the substance of encapsulation.

The member procedure `Square`, having the same name as the class, is a *constructor*. It specifies how a variable of type `Square` is to be initialized upon creation.

The following would be a trivial program using the class in all its important aspects:

```
#include <iostream.h>
#include <Square.h>
void main( void )
{
    Square s;
    cout << "Initial area: " << s.area() << endl;
    s.left(2.0); s.bottom(-3.0); s.width(6.2);
    cout << "Final area: " << s.area() << endl;
}
```

The first `include` loads a system package of input/output routines; the second `include` provides the declaration of class `Square` (assumed to be contained in a file named `Square.h`). “`cout <<`” provides printed output, and is defined by the `iostream` package. Input is provided by the same package through “`cin >>`.” The statement “`Square s;`” creates an instance of a `Square` by invoking the constructor. `s.area()` and `s.width(6.2)` illustrate the ways in which all member procedures (except constructors) are called.

A user who might attempt to circumvent the protection of `_width`, for example, is prevented from doing so. If the statement “`s._width = 6.2;`” had been written instead of “`s.width(6.2);`”, the compiler would have halted and reported an access violation.

Assume now that we wish to add rectangles to our language. In order to illustrate inheritance, we regard a rectangle as a kind of square that has a *height* in addition to a width:

```
#include <Square.h>
class Rectangle : public Square
{
    public:
        Rectangle( void )
            { _height = 1.0; }
        void height( double h )
            { _height = h; }
        double height( void )
            { return( _height ); }
        double area( void )
            { return( _height*_width ); }
    protected:
        double _height;
```

```
};
```

Class `Rectangle`, derived from `Square`, provides a revised definition of `area` and has added `_height` together with two member procedures that report and set the value of `_height`. The code that implements `area` for the `Rectangle` class can gain access to the protected member `_width` of `Square` by virtue of the derivation being declared public. General users of `Rectangle` are still prevented any access to `Rectangle`'s data members `_left`, `_bottom`, `_width`, and `_height`.

A trivial usage of rectangle would be as follows:

```
#include <iostream.h>
#include <Rectangle.h>
void main( void )
{
    Rectangle r;
    cout << "Initial area: " << r.area() << endl;
    r.left(2.0); r.bottom(-3.0); r.width(6.2); r.height(3.7);
    cout << "Final area: " << r.area() << endl;
}
```

Polymorphism is used to write programs that treat squares and rectangles as abstract objects with something in common; e.g., the ability to report their area. For example, a definition of the “>” operator that compared areas might be useful. To do this, a *base class* must be defined that specifies what commonality exists. For example:

```
class Shape
{
public:
    virtual double area( void )=0;
};
```

Here the `area` procedure's information (name, argument(s), return type) are listed, but no body of procedure code is given. Instead, the lack of an implementation for `area` is indicated by “=0.” The key word `virtual` indicates that the implementation for `area` may be supplied as appropriate from any descendant class.

A definition for the “>” operator might look as follows:

```
int operator>( Shape& shA, Shape& shB )
{
    if( shA.area() > shB.area() )
        { return( 1 ); } else { return( 0 ); }
}
```

(The use of “&” as a modifier to the arguments is a final technicality required by C++ in order to complete the polymorphic definition of the operator.)

Class `Square` can be made a derived class of `Shape` by changing the first line of its declaration from “`class Square`” to “`class Square : public`”

Shape” and including the file `Shape.h`, assumed to contain the declaration of `Shape` and the code for the comparison operator. `Rectangle` thereby becomes a descendant of `Shape`. Together they constitute an inheritance hierarchy: `Shape`⇒`Square`⇒`Rectangle`.

The following trivial main program would illustrate the use of the comparison operator just defined:

```
#include <iostream.h>
#include <Rectangle.h>
void main( void )
{
    Square s;
    Rectangle r;
    s.width(5.0);
    r.width(3.0); r.height(2.0);
    if( s>r )
        { cout << "Square is larger." << endl; }
    else
        { cout << "Rectangle is larger." << endl; }
}
```

(The inclusion of `Rectangle.h` causes a chain of inclusions for `Square` and `Shape`, making all declarations available to the program.)

§3. Spline Classes

Three inheritance hierarchies of C++ classes for use in the design and fitting of spline curves and surfaces at the University of Waterloo will be described. Each of the hierarchies was designed with polymorphism in mind to permit the implementation of general algorithms that could be used with different spline representations. The three hierarchies cooperate, each providing a collection of related services: the `FuncBasis` hierarchy, whose top classes abstract the features of basis function evaluation, the `Func` hierarchy, whose top classes abstract the assembly of basis functions into finished functions, and the `Refiner` hierarchy, whose top classes abstract the insertion of knots and, to a rudimentary extent, the conversion of spline representations.

Many of the algorithms used in the first and third hierarchies were influenced by work due to Barry and Goldman [2]. The elemental operations in that work are triads of the form

$$\mathcal{B}(\dots, u_n, \dots) := \alpha_{rs}(u_n, \tau_r, \tau_s)\mathcal{B}(\dots, \tau_r, \dots) + \beta_{rs}(u_n, \tau_r, \tau_s)\mathcal{B}(\dots, \tau_s, \dots)$$

involving the polar form (“blossom”), \mathcal{B} , of a spline. The choice of the affine (or sometimes linear) α ’s and β ’s represents the choice of the spline representation (basis), and the τ ’s are (or are sometimes related to) the knots.

The `FuncBasis` hierarchy uses triad schemes to produce all the nonzero basis values for any given segment of the domain. At the top of the hierarchy

is the `FuncBasis` class, which has virtual public member procedures `dimension` and `evaluate`. The member `dimension` is present to report the number of basis values that the member `evaluate` will produce. None of the member procedures is implemented, of course; the `FuncBasis` class is designed for polymorphic code that would accommodate such things as trigonometric functions as well as spline basis functions.

The `BBasis` class is derived from the `FuncBasis` class and has virtual member procedures `order`, `degree`, `numKnots`, `numBreakpoints`, `knot`, `breakpoint`, and `multiplicity`. A procedure such as `knot` allows the setting and reporting of values, but is unimplemented at this level of the hierarchy to permit specific variations on triad schemes to be implemented in derived classes.

At the lowest level of the hierarchy are several classes: currently `UBBasis`, `NUBBasis`, `BezBasis`, and `CMSBasis`, each individually derived from `BBasis`. In order they implement cardinal B-splines, general B-splines, Bézier splines, and connection matrix splines [4]. Each of these classes has member data related to knots: start and stride for the cardinal splines, knots themselves for the general B-splines, breakpoints for the Bézier splines, and finally, breakpoints with associated connection matrices for the CMS-splines. Each is responsible for implementing its own version of the `evaluate` procedure inherited from `FuncBasis` through `BBasis`.

`Func` is the top class of the `Func` hierarchy. Its main service, as far as the present discussion is concerned, is to provide an unimplemented virtual procedure `evaluate`. In contrast to the `FuncBasis` version, `Func`'s version of `evaluate` is intended to return a single value (or curve or surface point).

Derived from `Func` is `LCFunc`, to support the writing of polymorphic code for functions that are linear combinations of basis functions; e.g., trigonometric sums as well as splines. This class adds member procedures to set and report coefficients (or control vertices) and implements the `evaluate` member by combining coefficients with basis function values. In order to carry out the implementation, the class contains the coefficients as member data, but in order not to be tied to specifics, the basis functions are represented as member data only by pointers to objects of type `FuncBasis`. This means that the computation of the linear combinations that implements `evaluate` is polymorphic code itself. Some further details about an earlier version of the `Func` and `FuncBasis` hierarchies are provided in [1].

At the next level of the hierarchy comes the `BFunc` class, for which the pointers to `FuncBasis` objects in `LCFunc` are now reflected in `BBasis` pointers. Finally, at the lowest level, the `Func` hierarchy has classes `UBFunc`, `NUBFunc`, `BezFunc`, and `CMSFunc`, which provide spline objects of explicit representational type. In these classes, the member data that in the `BFunc` class consisted of pointers to `BBasis` objects is, for these classes, reflected in pointers to `UBBasis`, `NUBBasis`, `BezBasis`, or `CMSBasis` objects, as appropriate. With the `Func` hierarchy it is possible to compose code that works for any function, or any function that is a linear combination of basis functions, or any function

that is a linear combination of spline basis functions, or finally, functions that are specifically given in Bézier representation or the like.

The final hierarchy to be surveyed begins with the `Refiner` class. Refinement (*i.e.*, knot insertion) is implemented as a collection of classes that transform unrefined coefficients (control vertices) into refined ones based on the original knot sequence, the final knot sequence, and the spline representation. The `Refiner` class is intended to support polymorphic code that deals with refinement in general. This class has an unimplemented virtual procedure, `newCVs`, whose argument is the vector of the unrefined coefficients and whose return is the vector of the refined coefficients. The `Refiner` hierarchy is exceedingly shallow. Derived from the `Refiner` class is only a level of classes that implement the `newCVs` procedure as an appropriate instance of the refinement algorithms covered in [2]: `BezFastBreakInserter` for Barry-Goldman fast breakpoint insertion on Bézier splines, `BezRandomBreakInserter` and `BezSingleBreakInserter` for de Casteljau refinement, `NUBFastKnotInserter` for Barry-Goldman fast knot insertion on general B-splines, `NUBRandomKnotInserter` for the Oslo algorithm, and `NUBSingleKnotInserter` for Boehm's algorithm.

Finally, as a convenience to the user, the `BFunc` class and its descendants provide a member procedure, `addKnots`, that manages the entire transition from an unrefined basis with unrefined coefficients to a refined basis and refined coefficients. This procedure calls the appropriate member of the `Refiner` hierarchy to carry out its service.

The general flavor of code using these three class hierarchies is given in the examples below. In the first, a general B-spline basis is created from data read in; a B-spline function is generated; the function is evaluated and then refined. (Note that constructors can be written to require arguments.)

```
#include <iostream.h>
#include <Func/NUBFunc.h>
void main( void )
{
    unsigned order;
    double val, u;
    DoubleVec cvs;
    NumberSequence knots, addedknots;
    cin >> order >> knots >> cvs >> u >> addedknots;
    NUBBasis nb(order,knots);
    NUBFunc nf(cvs,nb);
    cout << "Original spline: " << nf << endl;
    cout << "First derivative: " << nf.evaluate(u,1) << endl;
    nf.addKnots(addedknots);
    cout << "Refined spline: " << nf << endl;
}
```

The `DoubleVec` class provides resizable arrays of doubles. The `NumberSequence` class is derived from the `DoubleVec` class by adding member procedures to keep the vector sorted. In the second example, neither basis nor function are created. The program simply uses a knot sequence and added knots to create a refinement object that is then used to transform unrefined spline coefficients into refined coefficients.

```
#include <iostream.h>
#include <Refiner/NUBRandomKnotInserter.h>
void main( void )
{
    unsigned order;
    DoubleVec cvs, newcvs;
    NumberSequence knots, addedknots;
    cin >> order >> knots >> cvs >> addedknots;
    NUBRandomKnotInserter nr(order,knots,addedknots);
    newcvs = nr.newCVs(cvs);
    cout << "Refined CV's: " << newcvs << endl;
}
```

References

1. Vermeulen, A. H., and R. H. Bartels, C++ splines classes for prototyping, *SPIE Vol. 1610: Curves and Surfaces in Computer Vision and Graphics II* (1991), 121–131.
2. Goldman, R. N., and T. Lyche (eds.), *Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces*, SIAM, Philadelphia, PA, 1993.
3. Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, New York, NY, 1988.
4. Seidel, H-P. Polar forms for geometrically continuous spline curves of arbitrary degree, *ACM Trans. Graphics* **12** (1993), 1–34.
5. Stroustrup, B., *The C++ Programming Language, second edition*, Addison-Wesley, Reading, MA, 1991.

Acknowledgements. This work has been supported through the Canadian Government's NSERC Strategic and Operating funding programs, through the Province of Ontario's ITRC funding program, and with the assistance of General Motors.

Richard H. Bartels
 Computer Science Department
 University of Waterloo
 Waterloo, Ontario N2L 3G1
 Canada
 rhbartel@ watcgl.uwaterloo.ca