Computer Assisted Mechanics and Engineering Sciences, 7: 185–193, 2000.

Object oriented programming and applications of boundary element method in ground vehicle aerodynamics[1]

Tak Wai Chiu
AEA Technology Rail,
rtc Business Park, London Road, Derby DE24 8YB, United Kingdom

(Received March 8, 1999)

The phenomenal development and popularisation of Object Oriented Programming (OOP) in recent years has created a new dimension in the innovation and implementation of powerful panel method techniques.

Although many scientists and engineers are already employing OOP languages such as C++, some of them are still using the languages in a procedural and non-hierarchical manner, leaving a large proportion of these languages' capability unexplored. This paper presents the idea and implementation of OOP in the panel method, which is widely used in ground vehicle aerodynamics. Program examples will show that OOP enables the writing of highly modularised, reusable, readable and debuggable panel method programs.

Keywords: object oriented programming, panel method, BEM, ground vehicle aerodynamics

1. Object oriented programming

Traditional programming methods look at a programme as essentially a process. A number of things, such as *variables*, are involved during the course of the process. Virtually everything is represented by a basic variable or a simple combination of basic variables: integers, real numbers, complex numbers, characters and booleans. The emphasis is usually placed on the process itself, while the variables are merely *things* that get involved in the process. For a simple process, a single (small) programme is written which is a simple transformation of the process from the beginning to the end. On the other hand, a lengthy process can usually be broken down into a number of modular processes: each of them is transformed into a subprogram. As the emphasis is placed on the process/procedure, this type of coding is usually called *procedural programming*. Fortran77 is a typical procedural programming language.

The becoming-fashionable Object Oriented Programming (OOP) technique, however, places the emphasis on the *things* (or, more precisely, the *objects*) involved in a process and how they interact with one another. The various elements of the process now become the member functions of the objects. The member functions of each object are analogous to the characteristics and behaviour of a physical object. In many physical applications, there are actual *things* involved and these automatically become good candidates as *objects* in OOP. In numerical applications, these objects can be nodes, finite elements, boundary elements, sub-grids, obstacles in an airflow, etc. Since the program involves physical objects with which human beings are familiar, they can easily be visualised and their interaction understood. It is also easier to model physical processes in a more physical way with OOP.

[1]Abbreviated version of this paper was presented at the VII Conference *Numerical Methods in Continuum Mechanics*, Stará Lesná, High Tatras, Slovakia, October 6–9, 1998, and published in its Proceedings.

With OOP, programs for numerical analyses/modelling can be written in a modular, reusable, readable and debuggable manner. This paper will show how this is the case.

2. Boundary element method (panel method) and Navier–Stokes solvers

Although in the last decade or so the Navier-Stokes solution methods have undergone rapid developments, the industrial use of the Panel Method in subsonic aerodynamics, and in particular, ground vehicle aerodynamics, has not been reduced. This is mainly due to the capability of the panel method in providing fast solutions over the object/vehicle concerned. Despite the fact that panel method, being a potential flow solution method, cannot model boundary layer separation directly, a lot of engineers find that it is an efficient pre-experiment design tool. For example, in the field of railway aerodynamics, the panel method is often used to investigate the effects of pressure waves created by the passing of high speed trains. The panel method can give the *potential flow* surface pressure distribution within a short period of time. Adverse pressure gradients, for instance, can be identified quickly, as possible locations of flow separation. This allows modifications to the design to be made in the early stage.

In the racing car sector, for instance, the designer is always subject to a very tight time scale. He/she will sometimes have only a day or two to modify a design or investigate a particular problem. The fast turn-around time of the panel method as compared to Navier-Stokes solvers helps to maintain its position in the industry and will continue to do so for years to come.

3. Panel method and OOP

The panel method [1–6, 8–9] and OOP are particularly good and obvious partners, mainly because the panel method is fundamentally object oriented – and these are not abstract objects, they are physically visualisable objects. There are physical objects everywhere in a panel method formulation: firstly there are nodes, nodes form edges, edges form panels, panels form parts of (or entire) obstacles in the air stream, and so on. The intrinsic hierarchical structure means that the programs are automatically modular and highly reusable. To illustrate the ideas that have been advocated here, we will build up a very simple program with C++. We will skip most of the details and will not worry too much about the syntax of this particular OOP language. For those who are interested in learning C++, reference [7] is one among the best of the books written on this programming language.

The panel method formulation we will assume for illustration here is developed in [2, 3, 5, 8, 9]. This particular formulation is especially efficient because it is based on a vectorial approach: the influence between panels (elements) is expressed in terms of vector equations based on the global co-ordinates. The first type of object we should discuss will therefore be *vector*. The purpose of these illustrations is to give the readers a feeling of the beauty of OOP.

3.1. Classes and objects

We must first understand the two terms: *class* and *object*. A class is the *type* or *identity* of one or more objects. An object is an instance of a class. For example,

- **Vector** is a class, while a particular vector **v1** is an object of class vector.
- **Motor Vehicle** is a class, while a **Transit Van** is an object of class Motor Vehicle.

We now use the definition of a vector type as a crash course in OOP. Most compilers, however, already provide the vector class as a *gift*. Note that in the example programs below, the lines/phrases beginning with '//' are *comments* and not part of the program statements.

```
void main()
{
    vector v1;  // a vector called v1 declared in the main program
    ⋮
}
```

C++ allows us to define the vector class such as below. A class has its data members and the functions then operate on the data members:

```
class vector()
{
private:
    double i, j, k;
      // 3 double precision data members: the 3 components of a vector
public:
    vector(double a=0, double b=0, double c=0): i(a),j(b),k(c) {}
      // the constructor
    double Length();
      // member functions which compute the length, or modulus, of the vector
    vector operator+(vector);  // defines the summation operator: +
    vector operator-(vector);  // defines the subtraction operator: -
    vector operator*(vector);  // defines the vector product operator: *
    double operator%(vector);  // defines the scalar product operator: %
    ⋮                          // there can be many other functions
    ⋮
};
```

In the above example, the constructor takes three values as arguments, say a, b and c, each of which has a default value of 0 if not provided. The constructor takes the values of a, b and c as the three components i, j and k. The other functions and operators, however, are only *declared* above. The actual bodies of these functions need to be defined elsewhere, which we will skip here. With the above we have defined the values, i.e. the three components, and the personalities, i.e. the member functions, of the `vector` class. The above class declaration will thus provide a variable type `vector` to be used in an analysis program, for instance:

```
void main()
{
    vector v1(1.2, 3.5, 4.8), v2(1.3, 3.8, 2.1), v3;
      // defines two vectors, v1=1.2i+3.5j+4.8k, etc and v3 has zero components using
      // the constructor
    double par1;      // defines a double precision variable named par1
    v3 = v1+v2;       // calculates v3 as the sum of v1 and v2
    par1 = v1%v2;     // calculates par1 as the scalar product of v1 and v2
    ⋮
}
```

We can also define more functions for the `vector` class to model some other possible types of behaviour/characteristics, such as the multiplication with a scalar or a matrix.

3.2. A generic panel (boundary element) class

Having gone through a very simple illustration in the previous example, we can see how we can define a generic panel type, such as:

```
class ThreeDPanel
{
protected:
    int     NoOfNodesInPanel;
    vector  **Node;   // pointer to an array of pointers to the nodes
    vector  *Edge;    // pointer to an array of edges
    vector  CollocationPoint;
    vector  NormalVector;
    double  Area;
    double  Perimeter;
public:
    ThreeDPanel();  // default constructor which does nothing
    ThreeDPanel(vector* node1, vector* node2, vector* node3,
      vector* node4 = NULL);
      // constructor with arguments: pointers to the node coordinates (either 3 or 4)
      // Argument 4 is optional, which is assumed to have a NULL value if omitted.
    vector GetCollocationPoint();
      // member function which returns the coordinates of the collocation point of the 3D panel
    vector GetNormalVector();
      // member function which returns the normal vector of a 3D panel
    double NeumannCondDueTo(vector FreeStream);
      // member function which computes the external Neumann boundary conditions
      // due to a freestream at the collocation point of 'this' panel
    :
};
```

Again the details of the bodies of the member functions, etc., are not given here. The constructor should be so defined that it will take the coordinates of the nodes and compute the edges, collocation points, normal vector, area and perimeter of the panel. For instance, somewhere in the main program, a statement like

```
ThreeDPanel pan1(node1, node2, node3, node4);
```

will make use of the constructor and declare a variable (object) named `pan1` as a `ThreeDPanel` and the edges, collocation points, normal vector, area and perimeter of `pan1` will be computed automatically as defined in the constructor.

3.3. A constant source panel class – class inheritance

We will never need to use a generic 3D panel as described in the previous section. If we consider the non-lifting potential flow over a three-dimensional body, we will need constant or linear source panels. If we consider the lifting flow over a wing, however, we will need, for instance, constant or linear doublet panels. Sometimes we may need a combination of these panels in a particular problem. However, no matter which type of panel we want to use, they share some very similar properties. They will all have nodes, edges, a collocation point, a normal vector, an area and a perimeter, for instance. It would be nice not to have to define each of these panel types from scratch.

Some OOP languages such as C++ offer the powerful tool of *class inheritance,* which allows us to produce new classes from existing classes by inheriting their properties. The class from which one or some other classes are derived is called the *base class*, while the classes inheriting its properties are called the *derived classes.*

```
class ConstantSourcePanel: public ThreeDPanel
{
protected:
```

```
    double *Strength;
public:
    ConstantSourcePanel();
    ConstantSourcePanel(double *S, vector* node1, vector* node2,
      vector* node3, vector* node4 = NULL);
      // Constructor for ConstantSourcePanel, which will make use of the contructor of
      // the ThreeDPanel class, and give the member variable 'Strength' the value 'S'
      // from the list of arguments
    double ContributesNeumannCondTo(ThreeDPanel *pan2);
      // member function which computes the contribution of pan2 to the Neumann Boundary
      // Condition at the collocation of 'this' panel. In other words, the function computes an
      // entry of the matrix on the left-hand side of the linear system to be solved.
    vector VectorialInfluenceCoefAt(vector* point1);
      // influence coefficient induced by 'this' panel at a point with position vector 'point1'
    vector VectorialInfluenceCoefAt(ThreeDPanel *pan2);
      // influence coefficient induced by 'this' panel at the collocation point of another panel, 'pan2'
    vector InducesVelocityAt(vector* point1);
      // velocity induced by 'this' panel at the point 'point1'. This is equal to the product
      // of the Vectorial Influence Coefficient and the source strength of the panel.
    vector InducesVelocityAt(ThreeDPanel *);
      // velocity induced by 'this' panel at the collocation point of another panel, 'pan2'
};
```

In the above example we derive the `ConstantSourcePanel` class from the `ThreeDPanel` class. This means that the derived class will automatically have all the class data members and member functions of the base class, plus some additional data members and functions as above. Similarly we can derive other panel types, e.g. doublet panel, from the `ThreeDPanel` class. It is also possible, if required, to derive classes from a derived class. In that case, `ThreeDPanel` will have grandchildren (Fig. 1)!

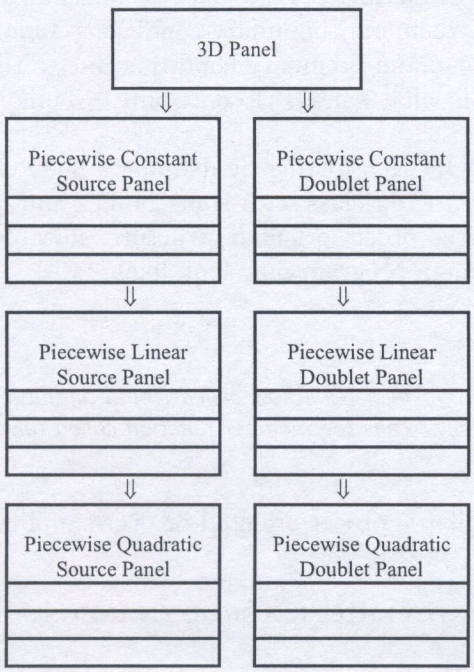

Fig. 1. A simple class hierarchy for panel method applications

3.4. What would the main program be like?

We can, and should indeed, continue to create higher level classes. For instance, a `Body` class can be defined such that each `Body` object will have a certain number of `ConstantSourcePanel` objects (or pointers to those objects) as its data members. The idea will be similar. But for now we will not further the hierarchy so that we can illustrate the interactions of the panels in a main program. When the classes are properly defined, the main program will be extremely easy to write, read and manage. For instance, in a program which deals with non-lifting potential flows, we can see statements like:

```
void main()
{
    ⋮
    ConstantSourcePanel** Panel = NewArray(Panel,TotalNumberOfPanels);
      // declares an array of pointers to ConstantSourcePanel of dimension equal to the
      // TotalNumberOfPanels: Panel[1] ... Panel[TotalNumberOfPanels]
    ⋮
    for(k=1;k<=TotalNumberOfPanels;k++)                              A
    {
      RHSV[k] = Panel[k]->NeumannCondDueTo(FreeStream);              B
      for(j=1;j<=TotalNumberOfPanels;j++)                            C1
      {
        LHSM[k][j] = Panel[j]->ContributesNeumannCondTo(Panel[k]);   C2
      }
    }
    ⋮
}
```

Within the double-loop beginning with line A, the system of linear equations is formed: `LHSM` being the left-hand-side-matrix containing the influence coefficients, `RHSV` being the right-hand-side-vector containing the (external Neumann) boundary conditions. Line B can actually be read as "the entry `RHSV[k]` is `Panel[k]`'s Neumann boundary condition due to the freestream". Line C2 can be read as "the entry `LHSM[k][j]` is what `Panel[j]` contributes Neumann boundary condition to (the collocation point of) `Panel[k]`".

One should of course take OOP even further by defining a `Body` class which will have 3D panels as its data members, and then a `Flow` class with some bodies and a `Freestream` as its members. This will create a highly organised object oriented structure, suitable for large scale CFD software development. In that case the main program may look like:

```
void main()
{
    Flow NonLiftingFlow;    // declares a flow called NonLiftingFlow
    NonLiftingflow.run();   // calls the member function called run
}
```

Whether this might look a bit too object-oriented or over-simplified is only a matter of taste.

4. Applications in Ground Vehicle Aerodynamics

In the applications of panel methods in ground vehicle aerodynamics, object orientation helps to make programs more logically organised when different parts of the vehicle body requires different types of panels. For instance, in the formulation presented in [2] and [3] for the flow over a train in a cross-wind, the train body consists of constant source panels while the wake is made up of

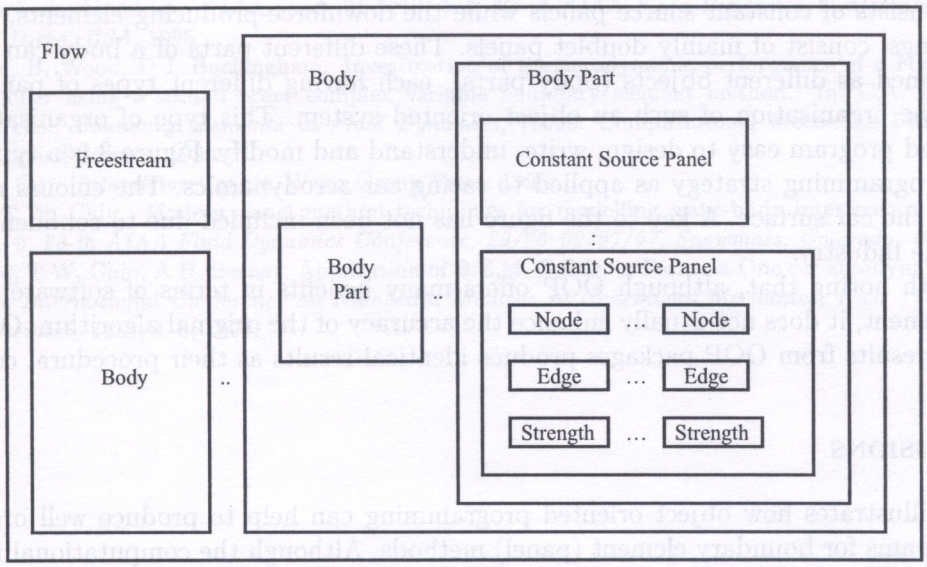

Fig. 2. An object ownership structure for panel method applications

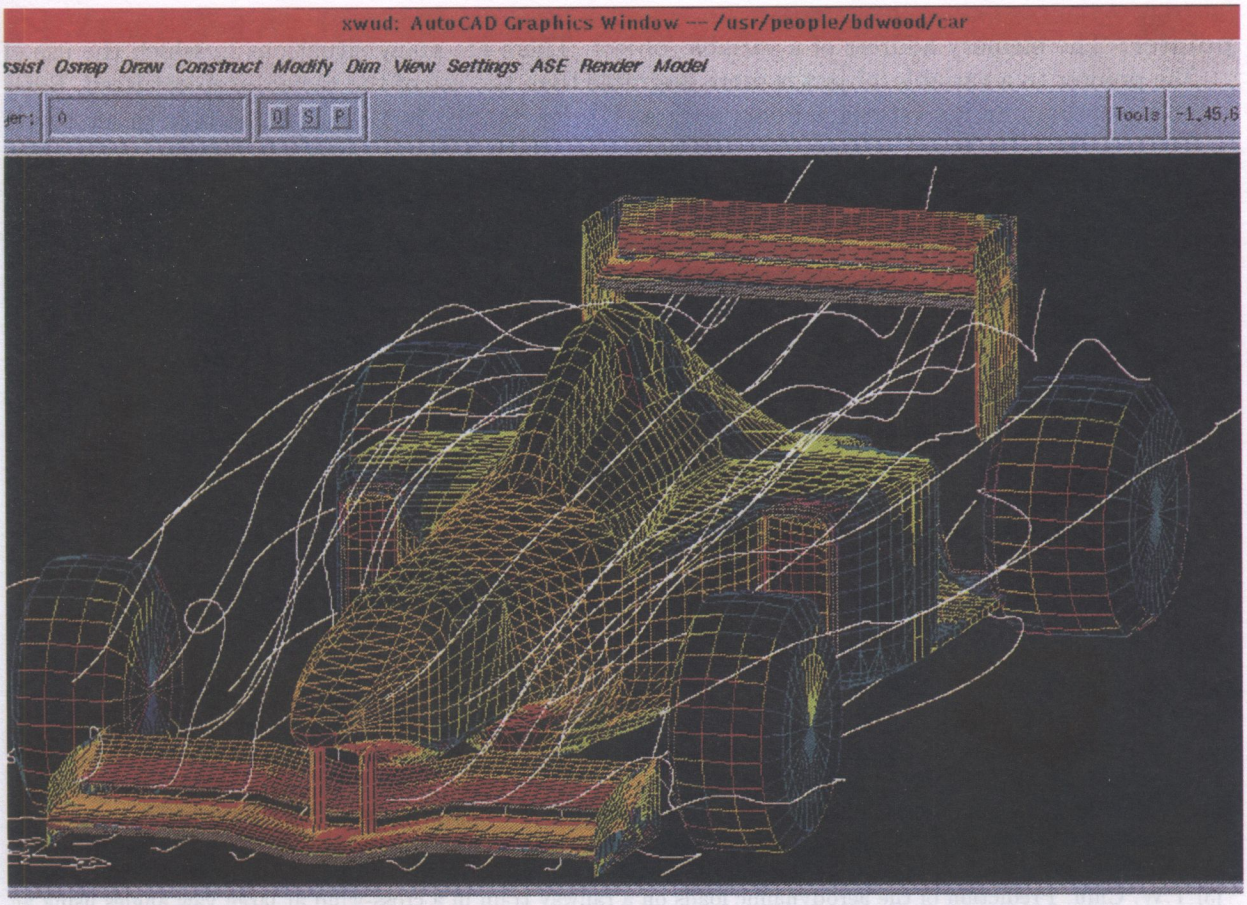

Fig. 3. An example of the result of an OOP panel method computation. The colours represent the pressure on the car surface. A key has not been included due to confidentiality

vortex panels. Similarly, in the formulation described in [2, 5, 8] for the flow over a racing car, the car body consists of constant source panels while the downforce-producing elements, i.e. the front and rear wings, consist of mainly doublet panels. These different parts of a body can conveniently be programmed as different objects (body parts), each having different types of panels. Figure 2 illustrates the organisation of such an object oriented system. This type of organisation makes a panel method program easy to design, write, understand and modify. Figure 3 is a typical example of such a programming strategy as applied to racing car aerodynamics. The colours represent the pressure on the car surface. A key to the figure has not been included due to confidentiality in the Formula One industry.

It is worth noting that, although OOP offers many benefits in terms of software development and management, it does not usually enhance the accuracy of the original algorithm. Our tests have shown that results from OOP packages produce identical results as their procedural counterparts.

5. Conclusions

This paper illustrates how object oriented programming can help to produce well organised high quality programs for boundary element (panel) methods. Although the computational fluid dynamics (CFD) scene has been dominated by procedural programming languages such as Fortran77 until now, the use of OOP will surely benefit present and future programmers for years to come.

6. Definition of some programming terms

pointer: The memory address of a variable/object. There are many situations in which the use of the pointer to a variable/object is more desirable than the use of the variable/object itself. Refer to any programming reference book, such as [7], for details. In C++, the pointer to an object `vec1` is given by `&vec1`. Conversely, the object to which a pointer pvec1 is pointing to is given by `*pvec1`.

data member: A data that belongs to an object, which is declared in the class definition. For instance, for a vector class, the data members are the three components, i, j, and k. The data member can be accessed, in C++, using the syntax <ObjectName>.<DataMemberName> such as `vec1.k` or <PointerToObject>-><DataMemberName> such as `pvec1->k`

member function: A function that is defined within a class, which controls how a object of the class operates on its data members and interacts with other objects. The C++ syntax for accessing member functions is similar to that for data members.

public, protected, private: Special keywords in some OOP languages which controls the accessibility of data members and member functions by derived classes and other classes. The detailed meanings of these keywords are not relevant in the discussions in this paper.

References

[1] T.W. Chiu. *Complex Variable Boundary Element Method for the Design of Multi-Aerofoil Wings, with NWING for Windows.* Computational Mechanics Publications, UK, 154 pages, April 1997.

[2] T.W. Chiu. The applications of modern panel methods to vehicle aerodynamics: racing car and trains. In: H. Schmitt, ed., *Flow of Incompressible Fluids at High Reynolds Numbers. Advances in Fluid Mechanics Series*, Chapter 2, 35–74. Computational Mechanics Publication, UK, May 1997.

[3] T.W. Chiu. Prediction of the aerodynamic loads on a railway train in a cross-wind at large yaw angles using an integrated two- and three-dimensional source/vortex panel method. *Journal of Wind Engineering and Industrial Aerodynamics*, **57**: 19–39, 1995.

[4] T.W. Chiu. A two-dimensional second order vortex panel method for the flow in a cross-wind over a train and other two-dimensional bluff bodies. *Journal of Wind Engineering and Industrial Aerodynamics*, **37**: 43–64, 1991.

[5] T.W. Chiu, C.A.M. Broers, B.D. Wood, A.H. Berney. The application of modern panel method techniques to sport aero/hydrodynamics. *AIAA paper* 95-2210, 1–9, *26-th AIAA Fluid Dynamics Conference, 19–22 June 1995, San Diego, USA*, 1995.

[6] T.W. Chiu, B. Wood, D.J. Buckingham. Investigation of the aerodynamic performance of a Formula I multi-aerofoil spoiler using a second order complex variable boundary element method. In: C.A. Brebbia, P.W. Partridge, eds., *Boundary Elements in Fluid Dynamics*, 75–90. Computational Mechanics Publications and Elsevier Applied Science, 1990.

[7] R. Lafore. *C++ Interactive Course*. Waite Group Press, 1996.

[8] A. Terzi, T.W. Chiu. Modern panel method techniques for modelling wake-body interference. *AIAA paper* 97-1829, 1–9, *28-th AIAA Fluid Dynamics Conference, 29/06–02/07/97, Snowmass, Colorado, USA*, 1997.

[9] B.D. Wood, T.W. Chiu, A.H. Berney. Application of B.E.M. C.F.D. in Formula-One car aerodynamics. *Proceedings of the International Conference on Numerical Methods in Continuum Mechanics, High Tatras, Slovakia, 19–22 September, 1994*, 1–8, 1994.