*Stanford Exploration Project, Report 61, January 13, 1998, pages 1–123*

*Stanford Exploration Project, Report 61, January 13, 1998, pages 1–123*

# Object-Oriented programming for seismic data

*Jean-Claude Dulac and David Nichols*[1]

***keywords:*** *not available*


## ABSTRACT

The intuitive appeal of object oriented programming is that it provides better concepts and tools to represent the natural structure of the real world as closely as possible. The advantages of this direct representation capability in programming and data modeling are (1) data abstraction or encapsulation, the program objects have a one to one correspondence with the problem objects, and (2) reusable toolkits with well-defined minimal interfaces. We have created classes of SepData objects for seismic data manipulation. Data are encapsulated into objects such as Cube, Plane and Trace. Graphic objects are attached to these data objects to facilitate interactive programming. This library is already used for an object-oriented implementation of the Zplane and Overlay programs.


## INTRODUCTION

Objects are abstract data types which have "an existence of their own" and can in some sense communicate between each other. From the point of view of either object-oriented design or programming, an object is some private data together with the operations on that data. Object-oriented programming addresses language, implementation, and programming environment issues. To arrive to an object oriented solution to a problem, we must define what the objects should be.

### Sources of objects

An object-oriented design is one in which each of the major modules corresponds to an entity either in the user's world or in an implementation of a solution to the user's problem. In seismic processing the program main entities are the data. These objects correspond to data stores. Many types of data may be needed such as Plane, Trace, Cube. Seismic operations are frequently represented in data flow diagrams. Each fragment consists of one process together with the input data, output data and control parameters. Frequently control parameters may become object candidate.

[1]**email:** not available

For example the dimension of the data set is an object containing axes definition. Furthermore the splitting between the object handling data manipulations and the object handling data coordinates simplifies the interfaces to the operation of the objects. Some of the data flows may also become candidate objects. For example, reading or writting the file and the header file may be the role of input and output objects. In the same way, the conversion from float to byte is a such complex function that an object called `ByteConverter` is needed. Furthermore this relationship object between a float data set and a byte data set makes it easy to use each of the objects separately. If the relationship had been implemented with internal operations extra functionalities and internal private data would have been necessary.

### Higher level structure

In a first approximation we might define only `Put` and `Get` functions on the data store objects. A more refined study shows that we want operations like `Window, Copy, Byte, Merge, Pad`. By defining some higher-level operations on many of the objects, we move most of the problems into the objects that are, by the same strategy, made more reusable. A question remains. How do we arrange these objects with the best kind of interconnection? The simplest is to have a traditional hierarchical structure. We can create a superobject called Data which will encapsulate the objects Cube, Plane and Trace and associated operations. This Data superobject corresponds to coherent options of the Cube, Plane or Trace objects.

The use of high level objects yields the same advantages as low-level objects: encapsulation and isolation of related data and operations, enhanced mapping between the user's view of the system, potential reusability of subsystems, and increased control of complexity. Methods given to the Data Object are the operations cited in the previous paragraph (`Window, ...`). These functions can be applied to `Data` objects (Plane, ...). without making any assumption on the type of object. The behavior and the implementation becomes then very similar to the seplib programs where, from the user's point of view, no consideration on the data type is made when using a program.

### Refinement

In contrast of the bottom-up analysis which let us create a superobject, we may want to separate the Data objects into disk data objects and memory data objects. The splitting will avoid an additional parameter at object creation time that specifies which one of these two objects we want. But we see also that there is a high connectivity between this two type of objects. In fact the Data object is not a data store object but a manipulator of the data. A data store object, or really two data store objects are needed. These objects will have only `Put` and `Get` functionalities. They will either work on disk or in memory. The Data object as a manipulator of the data will have the data store object as internal data.

| **SepData Object** | |
|---|---|
| *functions* | *comments* |
| SepData(SepData *) | Creates a sepData from another one. |
| SepData(SepInfo *, void *, int esize) | Creates a SepData from a data structure definition and from the array of data. |
| SepData(SepInput *) | Creates a SepData from a SepInput object. |
| void Pad(int padstart[], int padend[]) | Pads this object to a new size. |
| void Merge(SepData *) | Merges a data object with itself. |
| void Window(int first[], int jump[], int num[]) | Windows itself. |

Table 0.1: SepData object- base class for data object. The SepPlane, SepTrace, or SepCube objects have exactly the same operations.

## OBJECTS

### Seplib objects

The seismic objects were inspired by the seplib philosophy (Claerbout 1986). Axis parameters describe the data structure and data objects are responsible for basic manipulation such as Append or Remove data, Get and Put data. Tables 0.3, 0.4, 0.5 and 0.1 in appendix describe the basic functionalities of the seplib objects.

### Graphic objects

The object-oriented paradigm is more often used in the graphic domain, and one of the purpose of the creation of these `SepData` class was to be able to have equivalent graphic objects. We create classes of Seismic graphic objects such as `GSPlane` to display Plane Data, `GSDataPict`, `GSCurvesPict` to handle seismic coordinates views, or `GSMultiLine, GSPolygon, GSBSPline` which displays different graphic objects and provide 1D editing facilities. All these graphics objects output themselves into a vplot file (Cole and Dellinger 1989). All these graphic objects have been encapsulated into a generic graphic object to give the user a unique representation of all the common functionalities of these objects.

The coordinate system of a `GSDataPict` or a `GSCurvesPict` is defined in two steps: First, specification of the screen rectangle in which this picture will be placed. Second, specification of which axis goes where, i.e. if the 1-axis is along the vertical direction or if it runs up or down, left or right. All graphic objects inserted into this picture are defined in seismic or user coordinates. This object serves the same purpose as the window concept introduce by GI (Claerbout 1989).

`GSMultiLine, GSPolygon, GSBSPline` or generically `GSNPtObject` know how to create handles and modification shapes for editing purposes or reshape themselves

| **Basic Graphic Objects** | |
|---|---|
| *functions* | *comments* |
| GSObject(Graphic *) | Creates a new graphic object having a copy of the given graphic state. |
| void Hardcopy(SColorMap *, char *file, SepData *) | Generates a vplot hardcopy of the current graphic. |
| GSNPtObject(Graphic *) | Creates a multi points graphic object. |
| Rubberband * CreateRubberShape(Coord, Coord) | Creates, stores and returns a rubberband representing the GSNPtObject for the user to reshape. Selects the control point from the input Coordinates. |
| GSNPtObject * GetReshapedCopy() | Creates and returns a copy of the GSNPtObject incorporating the change made to its shape. |

Table 0.2: Graphic object - base class for graphic object. The base object provides a set of functionalities which encapsulate all operations useful in the derived class. This encapsulation enhances code reusability and code simplification by providing an homogeneous interface for all types of graphic objects.

after an editing operation. The code to reshape all types (`Trace`, `Rect`, ...) of `GSNPtObject` is presented in Figure 1.

```
oldObject = Select(e.x,e.y) ;  // Select the graphic under the cursor.
rubberVertex = oldObject-> CreateRubberShape(e.x,e.y); // selection's rubberband
do {
   Read(e) ;                       // Read an event
   rubberVertex-> Track(e.x,e.y) ; // Track the event
} while( e.eventType != UpEvent )
newObject = oldObject-> GetReshapedCopy(); // Construct new reshaped object
```

Figure 1: Code fragment for reshaping every type of `GSNPtObject`. This example shows code reusability.

Tables 0.2, 0.6 and 0.7 present the basic functionalities of the graphic objects.

## INTERACTIVE OBJECTS

The interactive objects are derived from InterViews interactors (Dulac et al. 1988). The `View` has functionalities such as zooming, drawing, handling user events, changing display parameters such as clip, tpow, raster or wiggle mode. The `IAxis` are "intelligent" axes which display the user current position inside the view. The axes are attached to the perspective of the view, such that the axes are always consistent with the part of the data being displayed. Table 0.8 in appendix presents the basic functionalities of the interactive objects.

## EXAMPLES

### Slice demo program

This program displays a cube, plane by plane. The frame counter is controlled by a vertical scrollbar.

```
main(int argc,char **argv) {
    sworld = new SWorld("graphics", argc, argv); // Seismic world

    rasCmap = new RasColorMap("IC",8,1) ; // Colormap (Intensity and Clip)
    sworld-> SetRasColorMap(rasCmap) ; // Set the colormap to the seismic world

    deck = new Deck(); // Deck or stack to insert all the data planes
    messageWindow = new HelpMessage("Help") ; // Message window to display coordinates

    sepInput = new SepInput("stdin") ; // Seismic objects
    gsInfo = new GSInfo(rasCmap) ;     // Graphic state constructed from the colormap
    for( int i=0 ; i < sepInput-> Axis(3)-> len ; i++ ) {
       sepPlane = new SepPlane(sepInput,3,i) ;   // create a sepPlane of index i

       gsPlane = new GSPlane(sepPlane,gsInfo) ;  // create a GSPlane from a sepPlane

       spict = new GSDataPict(gsPlane,0,0,500,500); // create a data picture

       grBlock = new View(spict,messageWindow);   // create the interactor object
       deck-> Insert(grBlock) ;   // Insert he view into the stack of planes

       delete sepPlane ;    // free float data representation.
    }

    hbox = new HBox(deck,new VBorder,new VScroller(deck)); // juxtapose deck and vertical scroller
    window = new VBox(hbox,new HBorder,messageWindow); // Construct the global window
    frame = new TitleFrame(new Banner("SEP","Cube demo","July 89"),window); // title frame
    sworld->InsertApplication(frame); // Put the window on screen.
    frame->Run();      // Run the application.
    delete frame;      // Delete the application.
}
```

Figure 2: slice demo program

The program constructs first an interactive seismic world with a certain colormap. Then it builds a stack or deck of data planes going through the data cube. Then it attaches the deck to a vertical scroller, inserts the window on the screen, and then waits for user manipulation. Figure **??** gives an overview of the program and Figure **??** shows a X-Window dump of the program.

**Zplane, Overlay**

Using this library, and SepView (Dulac 88), Zplane and Overlays have been rewritten in a object oriented manner. Thanks to object oriented programming, these programs use the same library of basic objects and *code reusability is a dream which has come true.* In the following we will give examples of how and where the reusability have taken place.

A `Manipulator` is responsible for a simple and particular task such as redigitization of the velocity curve in the program Overlay, construction of a box where the spectrum may be computed in the program Zplane. The separation in simple objects responsible for a task from the beginning to the end of the user manipulation helps to identify duplicate work and then avoids duplication. Furthermore it simplifies the implementation by suppressing all states variable and encourages reusability. For example, I have inserted in appendix a table presenting some Overlay's object operations (Table 0.9). An overview of the code to manipulate an overlay object is given in Figure .

```
hyperbola = new RubberHyperbola(p,c,seisTrans,planeType,tmax) ; // Construct
do {
    Read(e) ;                      // Read an event
    if( e.eventType == DownEvent ) hyperbola-> Fix() ; // Fix apex
    hyperbola-> Track(e.x,e.y) ;  // Track the event
} while( e.eventType != UpEvent )
hyperbola-> HyperbolaParam(t0,x0,vel) ; // Get parameters
```

Figure 3: Code fragment to manipulate an Overlay object, here a `RubberHyperbola`. This example shows how coding is simplify through the utilization of objects.

We should recall from the introduction that an object is only defined by its methods or operations. Each view displaying data has been derived from the `View` class described in an earlier section. By using the inheritance mechanism, modification of tpow, of the clip value, or of the display mode are immediate functionalities. A `View` for the program Zplane will have extra functionalities such as computing the spectrum on a portion of its data. A `View` displaying Nmo Data will know how to recompute and redisplay its data when the velocity curve change.

The Seplib objects give a nice interface for saving filtered data in the Zplane program or velocity curves in the Overlay program. The `GSObjects` provide hardcopy operations. Then nothing has to be (re)done to create a Vplot hardcopy function for these programs. `GSObject` 1D editing operations give an immediate implementation of a "velocity curve reshape" functionality. The utilization of SepView helps to construct the interface and provides facilities such as file browser, documentation, help on line. As an illustration of the possibilities provided by such environment, some X-Window dumps and Vplot hardcopies of Overlay and Zplane user sessions are presented here. (Figures **??**, **??**, **??**, **??**, and **??**).

## CONCLUSIONS

We have presented an overview of our object oriented library encapsulating seplib functionalities and interactive functionalities. We have shown with examples that the concept of object-oriented design has increased code reusability and gives access to a high level programming environment for interactive seismic application. Object (oriented) programming allows a more direct representation of the real world model in the code. The problem of encoding is greatly reduced. Code reductions ranging from 40 to 90 percent have been experienced.

## REFERENCES

Claerbout, J.F., 1986, Canonical program library: SEP–**50**, 281-289.

Claerbout J.F., 1989, Interface for system independent plotting: SEP–**60**, 391-412.

Cole S., Dellinger J., 1989, Vplot:SEP's plot language: SEP–**60**,349-365.

Dulac J.C., 1988, A user interface manager: SepView: SEP–**59**.

Dulac J.C., Nichols D., Van Trier J., 1988, An introduction to InterViews: SEP–**59**.

## APPENDIX A

First, the appendix presents an overview of the Seplib objects `SepInput, SepOutput` which represents respectively an input or ouput header file and data file. The classes `SepInfo, SepAxis` which describe the structure of the data are also presented. Second the graphic objects derived from the GSObject or GSNPtObject such as `GSPlane, GSMultiLine, GSBSplines` are described. Third we present some InterViews interactor `View, GraPanel, Iaxis` used into the examples. Then we describe in one final table the interface of two rubberbands used into the Overlay program and presented in the Figure , respectively `RubberHyperbola` and `RubberParabola`.

| **SepInput Object** | |
|---|---|
| *functions* | *comments* |
| SepInput(char *) | Creates an input object. |
| boolean Par(char *name, int &value) | Gets a parameter value from its name. |
| boolean Par(char *name, float &value) | |
| boolean Par(char *name, char *&value) | |
| boolean Sample(DataStore *, int i1, int i2, int i3) | Gets a sample pointed by the 3 indices, puts it into the given DataStore object, returns true if everything went ok. |
| boolean Trace(DataStore *,int a1,int i1,int a2,int i2) | Gets a trace perpendicular to the plane given by the axes a1 and a2, and at the position i1,i2 on these axes, puts it into the given DataStore object. |
| boolean Plane(DataStore *, int ax1, int i1) | Gets a plane perpendicular to the axis ax1 at the position i1 on this axis. |
| boolean Cube(DataStore *) | Gets a cube and puts it into the given DataStore object. |

Table 0.3: SepInput object- base class for reading input datas. Input auxiliary file or standard input file are not differentiated from the user point of view.

| **SepOutput Object** | |
|---|---|
| *functions* | *comments* |
| SepOutput(char *, SepData *) | Creates an output object on the given header file and with the given SepData. |
| SepOutput(char *, SepInput *, SepData *) | Creates an output object on the given header file and given SepData , copies the input header. |
| void Par(char *name, int value) | Appends a parameter into the header file on the form name=value. |
| void Par(char *name, float value) | |
| void Par(char *name, char *value) | |

Table 0.4: SepOutput object- base class for writting output datas. Output auxiliary file or standard output file have the same interface.

| **SepInfo and SepAxis Objects** | |
|---|---|
| *functions* | *comments* |
| SepInfo(SepAxis *[], int n, int esz, char *t) | Creates an info from axes definition. |
| SepInfo(SepInput *) | Creates an info from a header file. |
| SepAxis(SepInput *, int n) | Creates axis n from a header file. |
| SepAxis(float o, float d, int n, char *t) | Creates an axis from input parameters. |

Table 0.5: SepInfo object and SepAxis object - classes for data description.

| **GSPlane, GSMultiLine, . . . Objects** | |
|---|---|
| *functions* | *comments* |
| GSPlane(SepPlane *, boolean, Graphic *) | Creates a GSPlane from a SepPlane. |
| void SetRasterMode(boolean) | Sets the mode of representation. |
| GSPoint(float, float, Graphic *) | Creates a point. |
| GSLine(float, float, float, float, Graphic *) | Creates a line. |
| GSMultiLine(float *, float *, int, Graphic *) | Creates a set of connected lines. |
| GSPolygon(float *, float *, int, Graphic *) | Creates a polygon. |
| GSFillPolygon(float *, float *, int, Graphic *) | Creates a filled polygon. |
| GSBSpline(float *, float *, int, Graphic *) | Creates a BSpline specified<br>by the given control vertices. |

Table 0.6: Graphic plane object and other graphic objects - Class for representing a data plane object. The plane may be represented either in raster mode or in wiggle mode.

| **Picture Objects** | |
|---|---|
| *functions* | *comments* |
| GSDataPict(GSPlane *,<br>Coord x0, Coord y0, Coord x1, Coord y1,<br>int x=2, int y=-1, Graphic *g) | Creates a GSDataPict from a GSPlane<br>in a given rectangle<br>with a given orientation. |
| GSCurvesPict(SepPlane *,<br>Coord x0, Coord y0, Coord x1, Coord y1,<br>int x=2, int y=-1, Graphic *g) | Creates a GSCurvesPict from a SepPlane<br>in a given rectangle<br>with a given orientation. |
| GSCurvesPict(SepInfo *,<br>Coord x0, Coord y0, Coord x1, Coord y1,<br>int x=2, int y=-1, Graphic *g) | Creates a GSCurvesPict from user axes<br>definition in a given rectangle<br>with a given orientation (no data). |
| void Change(GSPlane *,int x=2,int y=-1) | Changes the data representation<br>into the same screen space. |
| void UserInput(Coord x, Coord y,<br>float &u1, float &u2, float &u3,<br>float &i1, float &i2, float &i3) | Transforms input coordinates<br>into user coordinates. |

Table 0.7: Graphic Data Picture - Displays a GSPlane at a certain location in the view with a certain orientation. This object holds the transformation from seismic coordinates to screen coordinates. All the GSObject described into seismic coordinates must be inserted into this picture.

| **Interactive Objects** | |
|---|---|
| *functions* | *comments* |
| View(GSObject *, Message *) | Creates a view around the gsobject. |
| void SetTpow(float tpow) | Sets tpow, recomputes and updates display. |
| void SetClip(float clip) | Sets the clip, recomputes and updates display. |
| void Raster() | Sets to raster mode and updates display. |
| void Wiggle() | Sets to wiggle mode and updates display. |
| void FrameIncr(int) | Gets the next plane and updates display. |
| void Load(char *) | Gets the new data set and updates display. |
| HIAxis(View*, Graphic *, Alignment a=Bottom, int size = 0) | Creates a horizontal axis associated with a view and a graphic. |
| VIAxis(View*, Graphic *, Alignment a=Left, int size = 0) | Creates a vertical axis associated with a view and a graphic. |
| GraPanel(View *, boolean axis, int scrollType) | Creates a graphic panel. |

Table 0.8: Graphic Interactors - classes to insert a graphic into a view controlled by scroller or panner, with or without axes.

| **Overlay Objects** | |
|---|---|
| *functions* | *comments* |
| RubberHyperbola(Painter *, Canvas *, Transformer *st, int pt, float tmax, float v=1.48) | Construct a RubberHyperbola with a seismic transformation a plane type(pt), an initial velocity(v). |
| void Param(float &t0, float &x0, float &v) | Get hyperbola parameters. |
| void Fix() | Fixes the apex of the hyperbola. The subsequent trackings will change the velocity. |
| RubberParabola(Painter *, Canvas *, Transformer *st, float o1, float d1, float o2, float d2, int n2, float * sloth) | Constructs a RubberParabola with a seismic transformation, seismic coordinates, a sloth vector. |
| void Param(float &t0, float &rms) | Gets parabola parameters. |
| void Fix() | Fixes the apex of the parabola. The subsequent trackings will change the rms velocity. |

Table 0.9: Overlay Objects - classes to draw Overlays. The encapsulation of all the state variables and private data leads to a clean interface which encourages reusability.