Chapter XXIV
Class Patterns and Templates in Software Design

Julio Sanchez
Minnesota State University, Mankato, USA

Maria P. Canton
South Central College, USA

ABSTRACT

This chapter describes the use of design patterns as reusable components in program design. The discussion includes the two core elements: the class diagram and examples implemented in code. The authors believe that although precanned patterns have been popular in the literature, it is the patterns that we personally create or adapt that are most useful. Only after gaining intimate familiarity with a particular class structure will we be able to use it in an application. In addition to the conventional treatment of class patterns, the discussion includes the notion of a class template. A template describes functionality and object relations within a single class, while patterns refer to structures of communicating and interacting classes. The class template fosters reusability by providing a guide in solving a specific implementation problem. The chapter includes several class templates that could be useful to the software developer.

DESIGN PATTERNS

Engineers and architects have reused design elements for many years (Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, & Angel, 1977); however, the notion of reusing elements of software design dates back only to the early 1990s. The work of Anderson (1990), Coplien (1992), and Beck and Johnson (1994) set the background for the book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (1995), which many considered the first comprehensive work on the subject.

The main justification for reusing program design components is based on the fact that the design stage is one of the most laborious and time-consuming phases of program development. Design reuse is founded in the assumption that once a programmer or programming group has found a class or object structure that solves a particular design problem, this pattern can then be reused in other projects, with

considerable savings in the design effort. Anyone who has participated in the development of a substantial software project appreciates the advantages of reusing program design components.

The present-day approach to design reuse is based on a model of class associations and relationships called a class pattern or an object model. In this sense, a pattern is a solution to a design problem. Therefore, a programming problem is at the origin of every pattern. From this assumption we deduce that a pattern must offer a viable solution; it must represent a class structure that can be readily coded in the language of choice.

The fact that a programming problem is at the root of every design pattern, and the assumption that the solution offered by a particular pattern must be readily implementable in code, are the premises on which we base our approach to this topic. In the context of this chapter we see a design pattern as consisting of two core elements: a class diagram and a coded example or template, fully implemented in code. Every working programmer knows how to take a piece of existing code and reengineer it to solve the problem at hand. However, snippets of code that may or may not compile correctly are more a tease than a real aide.

Although we consider that design patterns are a reasonable and practical methodology, we must also add that it is the patterns that we ourselves create, refine, or adapt that are the most useful. It is difficult to believe that we can design and code a program based on someone else's class diagrams. Program design and coding is a task too elaborate and complicated to be done by imitation or by proxy. A programmer must gain intimate familiarity with a particular class and object structure before committing to its adoption in a project. These thoughts lead to the conclusion that it is more important to explain how we can develop our own design patterns than to offer an extensive catalog of someone's class diagrams, which can be difficult to understand, and even more difficult to apply.

CLASS TEMPLATES

Occasionally, a programmer or program designer's need is not for a structure of communicating and interacting classes but for a description of the implementation of a specific functionality within a single class. In this case we can speak of a class template rather than of a pattern. The purpose of a class template is also to foster reusability by providing a specific guide for solving a particular implementation problem. In the following sections we include several class templates that could be useful to the practicing developer.

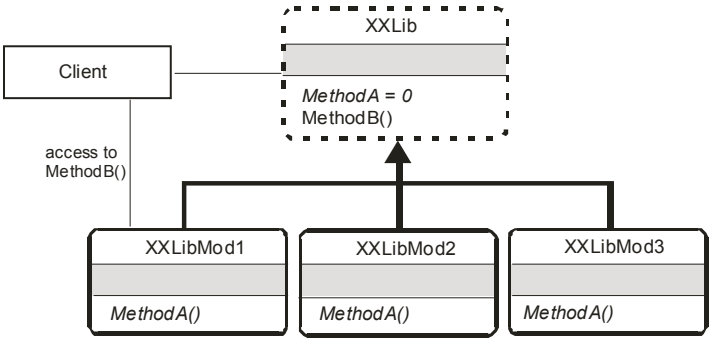
A Pattern is Born

We begin our discussion by following through the development of a design pattern, from the original problem, through a possible solution, to its implementation in code, and concluding in a general-purpose class diagram.

One of the most obvious and frequent uses of dynamic polymorphism is in the implementation of class libraries. The simplest usable architecture is by means of an abstract class and several modules in the form of derived classes that provide the specific implementations of the library's functionality. Client code accesses a polymorphic method in the base class and the corresponding implementation is selected according to the object referenced. But in the real world a library usually consists of more than one method. Since many languages allow mixing virtual and nonvirtual functions in an abstract class, it is possible to include nonvirtual methods along with virtual and pure virtual ones. The problem in this case is that abstract classes cannot be instantiated; therefore, client code cannot create an object through which it can access the nonvirtual methods in the base class. A possible but not very effective solution is to use one of the derived classes to access the nonvirtual methods in the base class. Figure 1 depicts this situation.

The first problem of the class diagram in Figure 1 is that the client code accesses the nonvirtual

Figure 1. A class library implemented through dynamic polymorphism



MethodB() in the base class by means of a pointer to one of the derived classes. A second, and perhaps more important one, is that method selection must take place in the client's code. Both of these characteristics expose the class structure and add a substantial processing burden to the client.

There are several possible solutions to the first problem. We could make the base class a concrete class with MethodA() as a simple virtual function, with no real implementation. In this case MethodB() becomes immediately accessible. Another solution would be to create a new class to hold the nonvirtual methods originally in the base class and have this new class inherit abstractly. However, neither of these solutions addresses the most important problem, which is that client code must perform method selection. It is this characteristic of inheritance that breaks encapsulation.

Encapsulation can be preserved by using object composition instead of inheritance. Also, combining object composition and inheritance achieves dynamic binding of polymorphic methods while preserving encapsulation. Figure 2 is a possible class diagram for implementing the library by means of object composition and inheritance.

Design of a VESA True Color Library

The design of a VESA true color graphics library can be based on combining object composition and class inheritance. The minimal functionality of this library can be stated as follows:

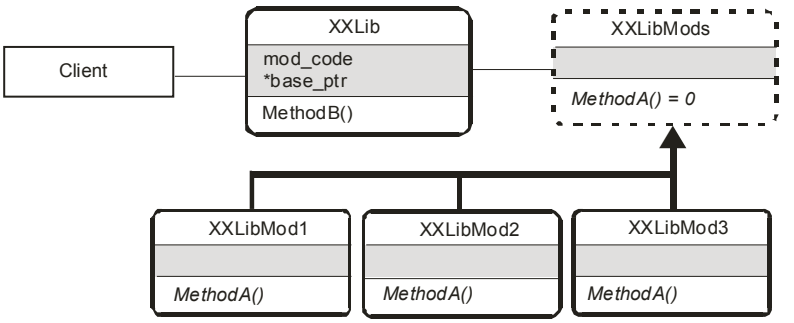
1. A way of setting the desired VESA true color mode.
2. A way of obtaining the current VESA mode as well as the vertical and horizontal resolution.
3. A way of drawing a screen pixel defined in terms of its x and y screen coordinates and its color attribute.
4. A way of reading the color attribute of a screen pixel located at given screen coordinates.

Figure 3 is a diagram of the library classes.

Observing the class diagram in Figure 3 we detect some features of the class structure:

1. The nonvirtual methods are located in the class named VesaTCLib.
2. VesaTCLib is a concrete class and can be instantiated by the client.
3. The mode-dependent, polymorphic methods for setting and reading pixels, named DrawPixel() and ReadPixel(), respectively, are located in an inheritance structure.
4. The methods named GetPixel() and SetPixel() in the class VesaTCLib provide access by means of a pointer to the polymorphic methods DrawPixel() and ReadPixel().

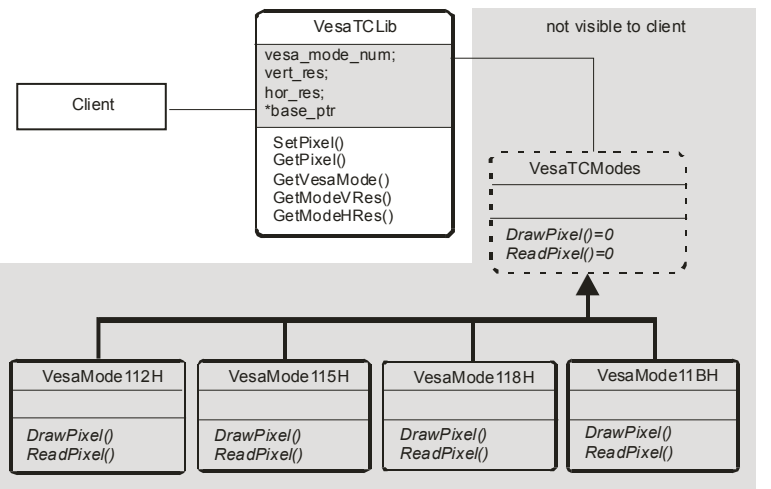
Figure 2. An alternative implementation of the class library



The result of this class design is that the implementation is now transparent to the client, as shown by the gray shaded background in Figure 3. The following program (Box 1) contains the schematic implementation of the class and inheritance structure in Figure 3.

The previous code sample, named SAMP-01.CPP, deserves a few additional comments. The constructor receives a coded signature that defines the VESA true color mode requested by the caller. A switch construct stores the VESA mode number, vertical and horizontal resolution, and the address of the library methods used in setting and reading a pixel in this mode. Therefore, the contingency code executes only once, when the object is created. Thereafter, each object of the class VesaTCLib is associated with a particular mode and keeps the address of the method to be used in its own pixel setting and reading operations. The methods GetPixel() and ReadPixel() of the class VesaTCLib are the client's interface to the pixel-level primitives in the library. The actual setting and reading of a screen pixel is performed as follows:

Figure 3. Class diagram for a VESA true color graphics library



Box 1.

```
//************************************************************
// C++ program to illustrate implementation of a VESA true
// color graphics library using object composition and class
// inheritance
// Filename: SAMP-01.CPP
//************************************************************

#include <iostream.h>

//************************************************************
//              classes
//************************************************************
// Abstract base class
class VesaTCModes {
public:
// Pure virtual functions
 virtual void DrawPixel(int, int, int) = 0;
 virtual unsigned long ReadPixel(int, int) = 0;
};
//*****************************
//  Polymorphic classes
//*****************************
// Note: methods have stub implementations in this demo
//   program

class VesaMode112H : public VesaTCModes {
public:
 virtual void DrawPixel(int row, int column, int color) {
 cout << "Setting pixel in Mode 112H\n";
 return;
 }
 virtual unsigned long ReadPixel(int row, int column) {
 cout << "Reading pixel in Mode 112H\n" ;
 return 0;
 }
};

class VesaMode115H : public VesaTCModes {
public:
 virtual void DrawPixel(int row, int column, int color) {
 cout << "Setting pixel in Mode 115H\n";
```

continued on following page

Box 1. continued

```
 return;
 }
 virtual unsigned long ReadPixel(int row, int column) {
 cout << "Reading pixel in Mode 115H\n" ;
 return 0;
 }
};
class VesaMode118H : public VesaTCModes {
public:
 virtual void DrawPixel(int row, int column, int color) {
 cout << "Setting pixel in Mode 118H\n";
 return;
 }
 virtual unsigned long ReadPixel(int row, int column) {
 cout << "Reading pixel in Mode 118H\n";
 return 0;
 }
};

class VesaMode11BH : public VesaTCModes {
public:
 virtual void DrawPixel(int row, int column, int color) {
 cout << "Setting pixel in Mode 11BH\n";
 return;
 }
 virtual unsigned long ReadPixel(int row, int column) {
 cout << "Reading pixel in Mode 11BH\n";
 return 0;
 }
};
//************************
// non-virtual classes
//************************
class VesaTCLib {
private:
 int vesa _ mode _ num;      // Object data
 int vert _ res;
 int hor _ res;

 VesaMode112H obj _ 112H;    // Objects of derived classes
 VesaMode115H obj _ 115H;    // required for filling pointer
 VesaMode118H obj _ 118H;
```

continued on following page

Box 1. continued

```
 VesaMode11BH obj _ 11BH;

 VesaTCModes *base _ ptr;  // Base class pointer
public:
 VesaTCLib (int);      // Constructor
// Other methods
 int GetVesaMode();
 int GetModeVRes();
 int GetModeHRes();
 void SetPixel(int, int, int);
 void GetPixel(int, int);
};
//*****************************
// Methods for class VesaTCLib
//*****************************
// Constructor
VesaTCLib::VesaTCLib(int vesa _ mode) {
/* The constructor is passed a mode code as follows:
  1 = VESA mode 112H
  2 = VESA mode 115H
  3 = VESA mode 118H
  4 = VESA mode 11BH
According to the mode selected, code sets the definition,
VESA mode number, and a pointer to the corresponding
object of the library module.
*/
 switch (vesa _ mode) {
 case (1):
  vesa _ mode _ num = 0x112;
  hor _ res = 640;
  vert _ res = 480;
  base _ ptr = &obj _ 112H;
  break;
 case (2):
  vesa _ mode _ num = 0x115;
  hor _ res = 800;
  vert _ res = 600;
  base _ ptr = &obj _ 115H;
  break;
 case (3):
  vesa _ mode _ num = 0x118;
  hor _ res = 1024;
```

continued on following page

Box 1. continued

```
    vert _ res = 768;
    base _ ptr = &obj _ 118H;
    break;
   case (4):
    vesa _ mode _ num = 0x11b;
    hor _ res = 1280;
    vert _ res = 1024;
    base _ ptr = &obj _ 11BH;
    break;
   default:
    vesa _ mode _ num = 0x0;
    hor _ res = 0;
    vert _ res = 0;
    base _ ptr = &obj _ 112H;
  }
}
// Methods for reading and setting a screen pixel
void VesaTCLib::SetPixel(int row, int col, int attribute) {
 base _ ptr->DrawPixel(row, col, attribute);
};
void VesaTCLib::GetPixel(int row, int col) {
 base _ ptr->ReadPixel(row, col);
};

// Methods that return the mode information
int VesaTCLib::GetVesaMode() {
 return vesa _ mode _ num;
}

int VesaTCLib::GetModeVRes() {
 return vert _ res;
}

int VesaTCLib::GetModeHRes() {
 return hor _ res;
}

//************************************************************
//             client code
//************************************************************

main() {
```

continued on following page

Box 1. continued

```
// Objects of class VesaTCLib
 VesaTCLib obj _ 1(1);    // Object and mode code
 VesaTCLib obj _ 2(2);
 VesaTCLib obj _ 3(3);
 VesaTCLib obj _ 4(4);

// Operations on obj _ 1, mode code 1
 cout << "\nVESA mode: " << hex << obj _ 1.GetVesaMode();
 cout << "\nHorizontal Res: " << dec << obj _ 1.GetModeHRes();
 cout << "\nVertical Res: " << obj _ 1.GetModeVRes() << "\n";
 obj _ 1.SetPixel(12, 18, 0xff00);
 obj _ 1.GetPixel(122, 133);
 cout << "\n";

// Operations on obj _ 2, mode code 2
 cout << "VESA mode: " << hex << obj _ 2.GetVesaMode();
 cout << "\nHorizontal Res: " << dec << obj _ 2.GetModeHRes();
 cout << "\nVertical Res: " << obj _ 2.GetModeVRes() << "\n";
 obj _ 2.SetPixel(12, 18, 0xff00);
 obj _ 2.GetPixel(122, 133);
 cout << "\n";

// Operations on obj _ 3, mode code 3
 cout << "VESA mode: " << hex << obj _ 3.GetVesaMode();
 cout << "\nHorizontal Res: " << dec << obj _ 3.GetModeHRes();
 cout << "\nVertical Res: " << obj _ 3.GetModeVRes() << "\n";
 obj _ 3.SetPixel(12, 18, 0xff00);
 obj _ 3.GetPixel(122, 133);
 cout << "\n";

// Operations on obj _ 4, mode code 4
 cout << "VESA mode: " << hex << obj _ 4.GetVesaMode();
 cout << "\nHorizontal Res: " << dec << obj _ 4.GetModeHRes();
 cout << "\nVertical Res: " << obj _ 4.GetModeVRes() << "\n";
 obj _ 4.SetPixel(12, 18, 0xff00);
 obj _ 4.GetPixel(122, 133);
 cout << "\n";

 return 0;
}
```

```
void VesaTCLib::SetPixel(int row, int col,
int atts) {
  base _ ptr->DrawPixel(row,  col,  attri-
bute);
};

void VesaTCLib::GetPixel(int row, int col)
{
 base _ ptr->ReadPixel(row, col);
};
```

The processing in this case is done with a smaller processing overhead. The principal advantage of the new class design and implementation can be summarized as follows:

1. Contingency code to select the corresponding mode-dependant, pixel-level primitives is now located in the constructor; therefore, it executes only when the object is created.
2. Client code does not need to perform the mode selection operations, which have been transferred to the library classes.
3. Client code does not see or access the class inheritance structure since the pixel-level operations are handled transparently.

Developing the Pattern

In the previous sections we addressed a programming problem and found one possible solution that could be implemented in code. We also constructed a class diagram that reflects the relationships and associations of this solution in object-oriented terms. In order to make this solution easier to reuse we can eliminate all the case-specific elements from both the pattern and the coded example. Furthermore, the resulting abstraction can be given a name that provides an easy reference to the particular case.

In selecting a name for a design pattern we must carefully consider its purpose and applicability. Observe that the class structure for constructing the VESA true color library is probably applicable to many programming problems that are not related to computer graphics, or even to libraries. Its fundamental purpose is to provide an interface to an inheritance structure so that its operational details are hidden from client code. Since interface is too general a term, we could think of the word concealer in this context. For the lack of a better term we call this pattern a concealer, since its purpose is to conceal an inheritance structure so that its existence and operation are made transparent to the client. Figure 4 shows the concealer pattern in a more abstract form.

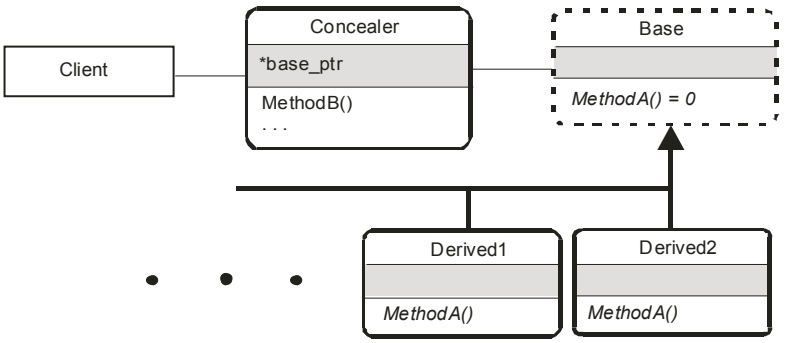
Unifying Dissimilar Interfaces

A simple but frequent design problem is to present a unified and friendly interface to a set of classes that perform different operations, for example, a set of classes that calculates the area of different geometrical figures, such as a parallelogram, a circle, a rectangle, and a square. The formula for the area of a parallelogram requires three parameters: the base, the side, and the included angle. The area of the circle and the square require a single parameter, in one case the radius and in the other one the side. The area of a rectangle requires two: the length and the width. Our task is to provide the client with a single interface to the calculation of the area of any one of these four geometrical figures. This class structure is known as a unifier pattern.

In the previous example the implementation could be based on the client passing four parameters. The first one is a signature code indicating the geometrical figure, the other three parameters hold the pertinent dimensional information. By convention, we agree that unnecessary parameters are set to NULL. The class diagram in Figure 5 represents one possible solution.

In the case of Figure 5 the method selection is based on an object of the corresponding class, therefore, it is a case of object composition. An alternative implementation could easily be based on pointers instead of object instances. Program SAMP-02.CPP shows the necessary processing in the first case (Box 2).

Figure 4. A Concealer pattern



Note that in program SAMP-02.CPP the objects are instantiated inside the switch construct in the GetArea() method. This ensures that only the necessary object is created in each iteration of GetArea(). Since the objects have local scope their lifetime expires when GetArea() returns. Therefore, only the necessary memory is used.

Unifier patterns have found frequent use in modeling object-oriented frameworks. InterViews 2.6 defined a similar construct for modeling interface elements such as buttons, scrollbars, and menus (Vlissides, 1988).

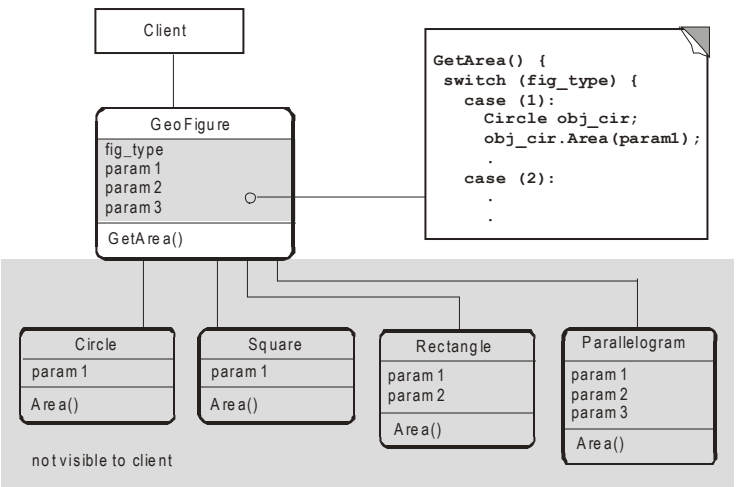
An Interface Pattern

By generalizing the class diagram in Figure 5 we can develop an interface pattern in which a class provides access to other classes that implement certain functionality. Figure 6 shows this generalization.

Aggregated Class Hierarchies

On occasions we may need to implement a set of classes related hierarchically. For example, a pro-

Figure 5. Implementing an interface class



Box 2.

```
//************************************************************
// C++ program to illustrate implementation of an interface
// class
// Filename: SAMP-02.CPP
//************************************************************

#include <iostream.h>
#include <math.h>
#define PI 3.1415

//***************************
//    classes
//***************************
//
class Circle {
public:
 float Area(float radius) {
  return (radius * radius * PI);
 }
};

class Rectangle {
public:
 float Area(float height, float width) {
  return (height * width);
 }
};

class Parallelogram {
public:
  float Area(float base, float side, float angle) {
   return (base * side * sin (angle) );
  }
};

class Square {
public:
  float Area(float side) {
   return (side * side);
  }
};
```

continued on following page

Box 2. continued

```
// Interface class
class GeoFigure {
private:
 int fig _ type;
 float param1;
 float param2;
 float param3;
public:
 GeoFigure(int, float, float, float);
 void GetArea();
};
// Constructor
GeoFigure::GeoFigure(int type, float data1, float data2,
      float data3) {
 param1 = data1;
 param2 = data2;
 param3 = data3;
 fig _ type = type;
};

// Implementation of GetArea() method
void GeoFigure::GetArea() {
float area;
 switch (fig _ type) {
  case (1):        // Circle
   Circle obj _ cir;
   area = obj _ cir.Area(param1);
   break;
  case (2):        // Rectangle
   Rectangle obj _ rec;
   area = obj _ rec.Area(param1, param2);
   break;
  case (3):        // Parallelogram
   Parallelogram obj _ par;
   area = obj _ par.Area(param1, param2, param3);
   break;
  case (4):        // Square
   Square obj _ sqr;
   area = obj _ sqr.Area(param1);
   break;
 }
```

continued on following page

Box 2. continued

```
   cout << "The area is of this object is: "
  << area << "\n";
};

//***************************
//     main()
//***************************
main() {
 GeoFigure obj1(1, 3, NULL, NULL);  // A circle object
 GeoFigure obj2(2, 12, 4, NULL);   // A rectangle object
 GeoFigure obj3(3, 12, 4, 0.7);    // A parallelogram object
 GeoFigure obj4(4, 3, NULL, NULL);  // A square object

 cout << "\nCalculating areas of objects...\n";
// Calculate areas
 obj1.GetArea();    // Area of circle object
 obj2.GetArea();    // Area of rectangle object
 obj3.GetArea();    // Area of parallelogram object
 obj4.GetArea();    // Area of square

 return 0;
}
```

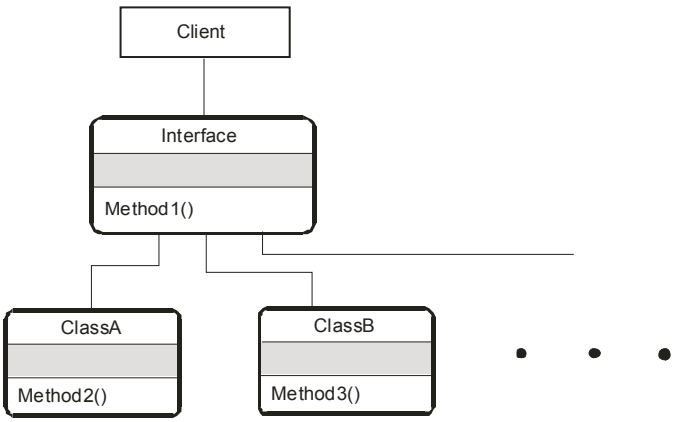
gram can draw geometrical figures, such as circles, squares, and triangles, where each of the figures is contained in an invisible, rectangular frame. Also assume that in this case the figure and its containing frame are so related that the creating of a figure (i.e., a circle, rectangle, or triangle) mandates the creation of its container frame.

A library of figure-drawing primitives could be provided by means of four classes: three to create the circle, square, and triangle figures, and one to generate the containing frame. The client would create an object of the corresponding figure and then one of the frame. Alternatively, figure-drawing classes could be linked to the frame generation class so that for each figure a corresponding frame object would be automatically created. With this approach, the frame generation operation is transparent to the client and programming is simplified.

We can add complications to the preceding example without altering its validity. For instance, we can assume that each frame implies the creation of another element called a border, and that the border must be contained in still another one called a window. Therefore, the resulting hierarchy is a geometrical figure, contained in a frame, requiring a border, which, in turn, must exist inside a window. In this example the object structure is mandated by the problem description since all the classes in the hierarchy are obligatory.

A class hierarchy can be implemented in C++ by inheritance, because the creation of an object of a derived class forces the creation of one of its parent class. If the inheritance hierarchy consists of more than one class, then the constructor of the classes higher in the hierarchy are called in order

Figure 6. An interface pattern



of derivation. Destructors are called in reverse order. The program SAMP-03.CPP demonstrates constructors and destructors in an inheritance hierarchy (Box 3).

When program SAM12-03.CPP executes the following messages are displayed:

```
Constructing a BaseA object
Constructing a Derived1 object
Constructing a Derived2 object
Destroying a Derived2 object
Destroying a Derived1 object
Destroying a BaseA object
```

A Class Hierarchy by Object Composition

One way of solving the problem described at the beginning of this section is to use inheritance. For example, we could implement a class hierarchy in which Circle, Square, and Rectangle were derived from a base class called Figure. The Figure class could contain polymorphic methods for calculating and drawing the geometrical object, as well as methods for creating the frame, the border, and the window. In this case the client would have the responsibility of calling all of the required methods. This is an example of how inheritance often constraints programming and exposes the underlying class structure.

Although in some cases a solution based on class inheritance may be acceptable, it often happens that the hierarchy of super classes is related to a particular method of a derived class, or to several related methods, but not to all of them (Gamma et al., 1995). Consider the case of several types of geometrical figures, all of which must be part of a window, contain a border, and be enclosed in a frame as described previously. In this case the figure-drawing methods could be made part of an inheritance structure; however, the methods that produce the window, border, and frame need not be part of the inheritance construct since these elements are required for any of the geometrical figures. One possible solution is to implement a class structure in which some methods form part of an inheritance association while others are implemented by means of object composition. Figure 7 shows a possible class diagram for this case.

In Figure 7 there are two mechanisms collaborating within the class structure. An inheritance element provides polymorphic methods that can be selected at run time in the conventional manner. Simultaneously, another class hierarchy is based on object composition. Note that the method Draw() of the classes Circle, Square, and Triangle, contains

Box 3.

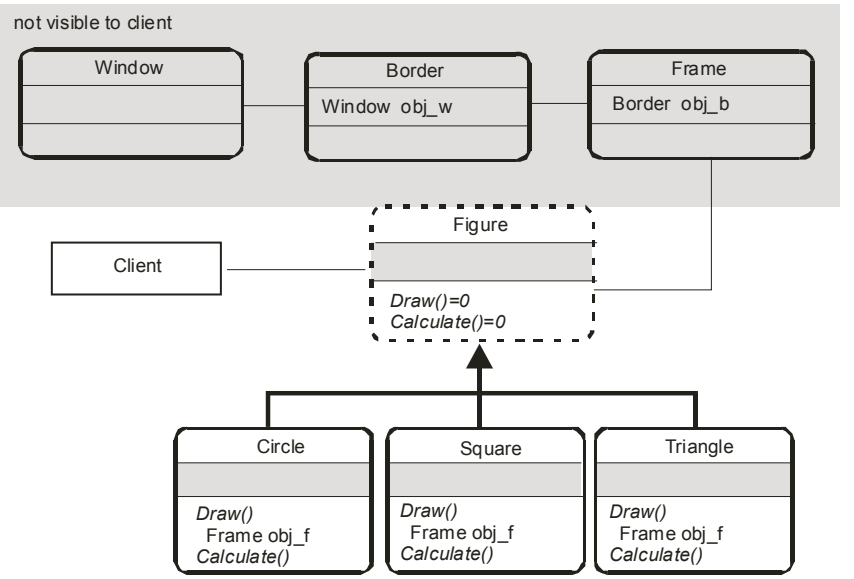
```
//************************************************************
// C++ program to illustrate constructors and destructors in
// a class inheritance hierarchy
// Filename: SAMP-03.CPP
//************************************************************

#include <iostream.h>

//***************************
//    classes
//***************************
class BaseA {
public:
 BaseA() {
  cout << "Constructing a BaseA object\n";
 }
 ~BaseA() {
  cout << "Destroying a BaseA object\n";
 }
};

class Derived1 : public BaseA {
public:
 Derived1() {
 cout << "Constructing a Derived1 object\n";
 }
 ~Derived1() {
 cout << "Destroying a Derived1 object\n";
 }

};
class Derived2 : public Derived1 {
public:
 Derived2() {
 cout << "Constructing a Derived2 object\n";
 }
 ~Derived2() {
 cout << "Destroying a Derived2 object\n";
 }
};
```

continued on following page

Box 3. continued

```
//*****************************
//      main()
//*****************************
main() {

// Program creates a single object of the lower class in the
// hierarchy. Constructors and destructors of the class higher
// in the hierarchy are automatically executed.

  Derived2 obj _ d;
  return 0;
}
```

Figure 7. Class hierarchy by object composition



an object of the class Frame. Also note that the class Frame contains an object of Border, which contains an object of Window. Therefore, we can say that a circle, a square, and a triangle are a kind of figure and that all of them have a frame, a border, and a window. The program SAMP-04.CPP is an implementation of the class diagram in Figure 7 (Box 4).

Note in the program SAMP-04.CPP, as well as in Figure 7, that it is the method named Draw() in

the concrete classes of the inheritance structure that instantiates the object of the class higher in the hierarchy, in this case the class named Frame. Once this object is referenced, the remainder of the hierarchy is produced automatically by means of the member object mechanism. The purpose of this construct is that the object hierarchy is generated when the method named Draw() executes, not when an object of the lower classes is instantiated. We can certify this operation when the program executes.

There may be cases in which we prefer that the hierarchy of super classes be instantiated at the time that the object of the lower class is created. In the example in Figure 7 this could be achieved by having a member object of the Frame class in the base class named Figure.

A Chain Reaction Pattern

In the class diagram of Figure 7 we note that when the lower classes (Circle, Square, and Triangle) instantiate an object of the class higher in the hierarchy, they start a chain reaction that produces objects of the entire hierarchy. We can abstract this operation by means of a class diagram that focuses exclusively on the chain reaction element, as is the case in Figure 8.

In Figure 8 we have implemented chaining by referencing the first chained object within a method

Box 4.

```
//*************************************************************
// C++ program to illustrate a class hierarchy by object
// composition
// Filename: SAMP-04.CPP
//*************************************************************

#include <iostream.h>

//***************************
//    classes
//***************************
class Window {
public:
Window() {
 cout << "Creating a window\n";
 }
};

class Border {
private:
 Window obj _ w;      // Border class contains Window object
public:
 Border() {
  cout << "Drawing a border\n";
```

continued on following page

Box 4. continued

```
  }
};

class Frame {
private:
 Border obj _ b;      // Frame class contains Border object
public:
 Frame() {
  cout << "Drawing a frame\n";
 }
 virtual void Draw() { return; };
};

// Abstract class
class Figure {
public:
 virtual void Draw() = 0;
 virtual void Calculate() = 0;
};

// Circle, Triangle and Square are at the bottom of the class
// hierarchy
class Circle : public Figure {
public:
 virtual void Draw() {
 Frame obj _ f;
 cout << "Drawing a circle\n";
 }
 virtual void Calculate() {
 cout << "Calculating a circle\n";
 }
};

class Square : public Figure {
public:
 virtual void Draw() {
 Frame obj _ f;
 cout << "Drawing a square\n";
 }
 virtual void Calculate() {
 cout << "Calculating a square\n";
 }
```

continued on following page

Box 4. continued

```
};

class Triangle : public Figure {
public:
 virtual void Draw() {
 Frame obj _ f;
 cout << "Drawing a triangle\n";
 }
 virtual void Calculate() {
 cout << "Calculating a triangle\n";
 }
};

//****************************
//     main()
//****************************
main() {


  Figure *base _ ptr;   // Pointer to base class
  Circle  obj _ c;   // Circle, Square, and Triangle
  Square  obj _ s;   // objects
  Triangle obj _ t;

  cout << "\n\n";
  base _ ptr = &obj _ c;  // Draw a circle and its hierarchical
  base _ ptr->Draw();   // super classes

  cout << "\n";
  base _ ptr = &obj _ s;  // Draw a square and its hierarchical
  base _ ptr->Draw();   // super classes

  cout << "\n";
  base _ ptr = &obj _ t;  // Draw a triangle and its hierarchical
  base _ ptr->Draw();   // super classes

  cout << "\n";
  base _ ptr->Calculate(); // Calculate() method does not generate
           // an object hierarchy
  return 0;
}
```

of the Chainer class, and then by member objects of the chained classes. There are many other ways of implementing a chain reaction effect.

Object Chaining

A programming problem often encountered consists of determining which, if any, among a possible set of actions has taken place. For example, an error handling routine posts the corresponding messages and directs execution according to the value of an error code. A common way of performing the method selection is by contingency code that examines the error code and determines the corresponding action. In C++ this type of selection is usually based on a switch construct or on a series of nested if statements. Alternatively, we can use object composition to create a chain in which each object examines a code operand passed to it. If it corresponds to the one mapped to its own action, then the object performs the corresponding operation, if not, it passes along the request to the next object in a chain. The last object in the chain returns a special value if no valid handler is found.

One of the advantages of using an object chain is that it can be expanded simply by inserting new object handlers anywhere along its members. To expand a selection mechanism based on contingency code we usually have to modify the selecting method by recoding the switch or nested if statements.

An Object Chain Example

A slightly more complicated case is one in which the selected object must return a value back to the original caller. For example, we define a series of classes that perform arithmetic operations which take two operands. The classes are called Addition, Subtraction, Multiplication, and Division. An operation code is furnished as a third parameter to a class called Operation, containing a method called SelectOp() that calls the Add method in the first object in the chain. In this case the object is of the Addition class. Add() examines the opcode operand; if it corresponds to the add operation, add executes and returns the sum to the caller. If not, it passes the object to Subtract, which proceeds in a similar fashion. Figure 9 shows the class structure for this example.

The program SAMP-05.CPP shows the processing details for implementing an object chain (Box 5).

An Object Chain Pattern

We can generalize the example of an object chain in Figure 9 and abstract its basic components. In this case the fundamental characteristic of the class structure is a series of related methods, each one of which inspects a program condition that determines whether the method is to respond with a processing

Figure 8. A chain reaction pattern

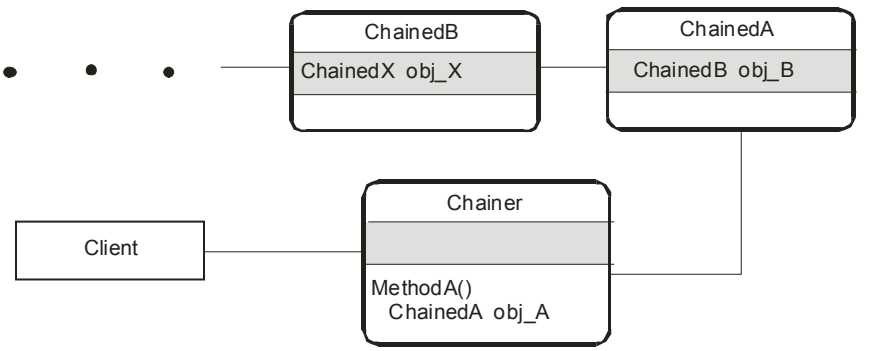
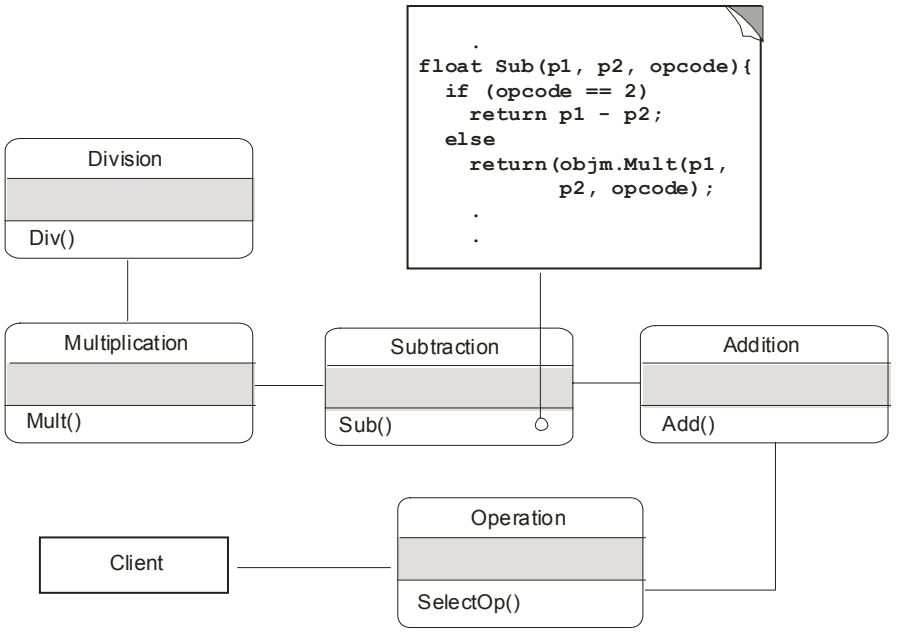


Figure 9. Class diagram for an object chain



action or pass the request along to the next object in the chain. Figure 10 is a class diagram for an object chain pattern.

STRING HANDLING CLASS TEMPLATE

The solution of some program development problems is based on patterns of interacting classes and objects, while others simply require a description of the internal structure of a single class. In this case we speak of a class template. For example, programs that deal with strings or that perform substantial string manipulations could profit from a particular class design that is optimized for string handling.

String Operations

A string that is defined at run time is sometimes difficult to store as an array since its length may not be known beforehand. The C++ new and delete operators can be used to allocate memory in the free store area, but it is easier to implement a class that performs all string-handling operations consistently and efficiently, rather than to create and delete each string individually. Implementing a string-handling class is possible because the new and delete operators can be used from within member functions and pointers are valid class members.

String operations often require knowing the string's length. The strlen() function defined in the string.h header file returns this value. Alternatively, we can implement a string as an object that contains a pointer to a buffer that holds the string itself and a variable that represents its length. A parameterized constructor can take care of initializing the string object storage using the new operator as well as its length parameter. The contents of the string passed by the caller are then copied into its allocated storage. This operation determines that the two data members associated with each string object are stored independently, however, they remain associated to the object and can be readily accessed as a pair.

In addition to the parameterized constructor, the proposed class could have a default constructor that

Box 5.

```
//************************************************************
// C++ program to illustrate an object chain
// Filename: SAMP-05.CPP
//************************************************************

#include <iostream.h>

//***************************
//    classes
//***************************
class Division {
public:
 float Div(float param1, float param2, int opcode) {
 if (opcode == 4)
  return (param1 / param2);
 else
   return 0;
 }
};

class Multiplication {
private:
 Division obj _ div;
public:
 float Mult(float param1, float param2, int opcode) {
 if (opcode == 3)
  return (param1 * param2);
 else
  return (obj _ div.Div(param1, param2, opcode));
 }
};


class Subtraction {
private:
 Multiplication obj _ mult;
public:
 float Sub(float param1, float param2, int opcode) {
 if (opcode == 2)
  return (param1 - param2);
 else
```

continued on following page

Box 5. continued

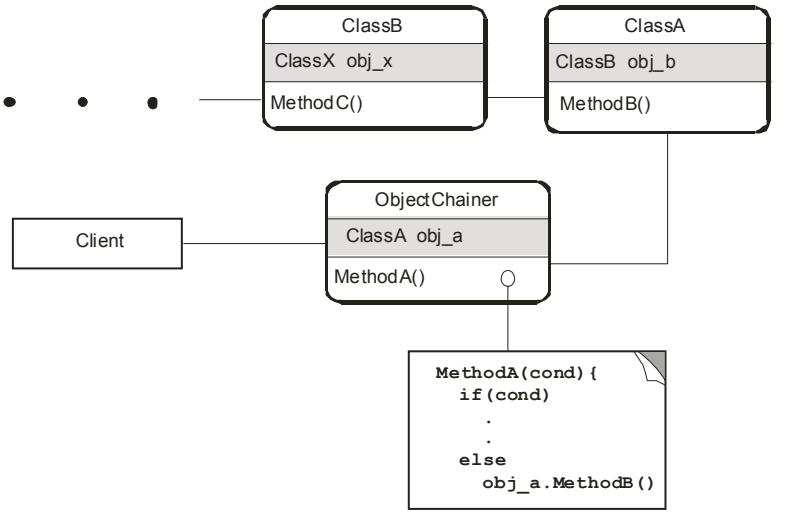
```
   return (obj _ mult.Mult(param1, param2, opcode));
  }
};

class Addition {
private:
Subtraction obj _ sub;
public:
 float Add(float param1, float param2, int opcode) {
 if (opcode == 1)
  return (param1 + param2);
 else
  return (obj _ sub.Sub(param1, param2, opcode));
 }
};

class Operation{
private:
 float param1;
 float param2;
 int opcode;
 Addition obj _ add;
public:
 Operation(float val1, float val2, int op) {
  param1 = val1;
  param2 = val2;
  opcode = op;
 }
 float SelectOp() {
 return (obj _ add.Add(param1, param2, opcode));
 }
};

//***************************
//     main()
//***************************
main() {
 Operation obj _ 1(12, 6, 1); // Declaring objects of the
 Operation obj _ 2(12, 6, 2); // four established opcodes
 Operation obj _ 3(12, 6, 3);
 Operation obj _ 4(12, 6, 4);
```

continued on following page

Box 5. continued

```
  cout << "\n";
// Performing operation on objects
  cout << "Operation on obj _ 1: " << obj _ 1.SelectOp() << "\n";
  cout << "Operation on obj _ 2: " << obj _ 2.SelectOp() << "\n";
  cout << "Operation on obj _ 3: " << obj _ 3.SelectOp() << "\n";
  cout << "Operation on obj _ 4: " << obj _ 4.SelectOp() << "\n";

  return 0;
}
```

Figure 10. Object chain pattern



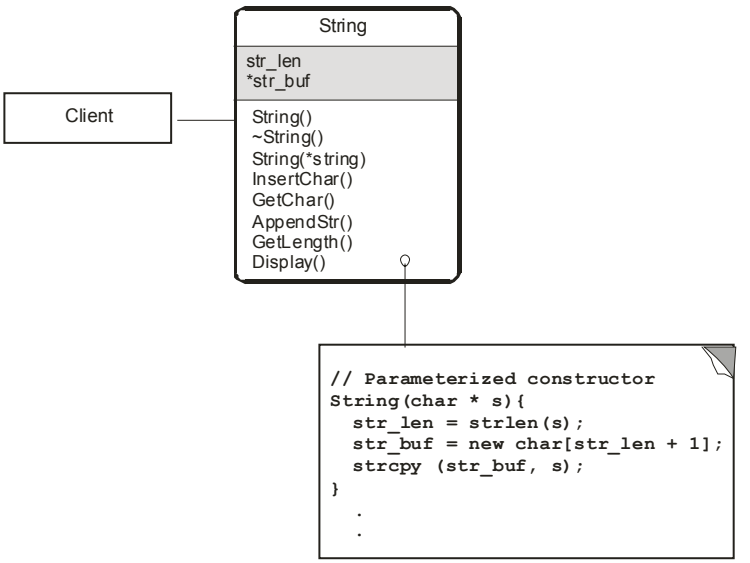
initializes both data members to zero whenever it is necessary. An explicit destructor method is also required in this case. The fact that the new operator is used to allocate space for the string buffer implies that the delete operator must be used to free the allocated memory. Other useful methods would be to get the length of the string and to insert a character in a string, to read a string character by specifying its index, and to append a new string to an existing one. Additional functionalities can be added by editing the class or by inheriting its methods. Figure 11 class diagram can serve as a template in this case.

The following program (Box 6) implements the string handling class template of Figure 11.

COMBINING FUNCTIONALITIES

In the implementation of libraries, toolkits, and application frameworks (Deutsch, 1989) we often come across a situation in which a given functionality is scattered among several classes. Rather than giving a client access to each one of these individual classes it is often a reasonable alternative to combine several methods into a single class

Figure 11. String handler class template



which can then be presented and documented as standard interface.

A Mixer Pattern

One of the practical uses of multiple inheritance is in combining functionalities by creating a class that inherits from two or more classes. The inheriting class serves to mix and unify the methods of the participating base classes. The class in Figure 12 shows a pattern based on multiple inheritance into a mixer class.

The following code fragment shows how to implement multiple inheritance in the case of the mixer class in Figure 12:

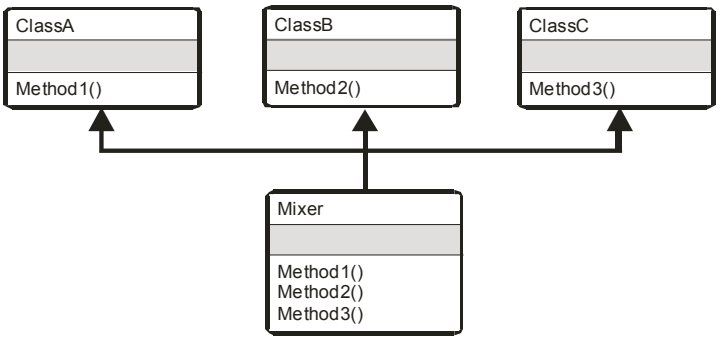
```
// Multiple inheritance
class Mixer : public ClassA, public ClassB,
public ClassC {
 // Implementation
};
```

When implementing multiple inheritance we must be careful to avoid situations which could lead to resolution conflicts; for example, inheriting a method that has the same name and interface in the parent classes, or inheriting multiple copies of the same method.

AN OBJECT-CLASSIFIER TEMPLATE

The objects in a class need not have identical signatures. A class can contain several parameterized constructors that create objects with different numbers or types of parameters. In this sense the constructors serve as object classifiers since each constructor executes according to the object's signature. The constructor can also store the object's type in a variable so that the object can be declassified during processing. For example, we wish to provide a class that calculates the area of various types of geometrical figures. Some figures such as the square require a single parameter, other figures such as the rectangle, have two parameters, and still others like the parallelogram have three parameters. If there is a parameterized constructor for each ob-

Figure 12. Mixer pattern for combining disperse functionalities



Box 6.

```
//*************************************************************
// C++ program template for string operations
// Filename: SAMP-06.CPP
//*************************************************************

#include <iostream.h>
#include <string.h>

//***************************
//    classes
//***************************
// String-handling class template
class String {
private:
 int str _ length;      // Length of string
 char *str _ buf;       // Pointer to string buffer
public:
// Declared constructors
 String();            // Default constructor
 String(const char *str);    // Parameterized constructor
// String processing methods
 void InsertChar(int index, char new _ char);
 char GetChar(int index) const;
 void AppendStr( const char *sub _ str);
 ~String();         // Explicit destructor
// Methods expanded in line
```

continued on following page

Box 6. continued

```
 int GetLength() const { return str _ length; }
 void Display() const { cout << str _ buf; }
};

// Implementation of the default constructor
String :: String() {
 str _ buf = 0;
 str _ length = 0;
}

// Implementation of the parameterized constructor
String :: String (const char *s) {
 str _ length = strlen(s);
 str _ buf = new char[str _ length + 1];
 strcpy( str _ buf, s);
}

// Implementation of operational methods
void String :: InsertChar(int index, char new _ char) {
 if ((index > 0) && (index <= str _ length))
  str _ buf[index - 1] = new _ char;
}
char String :: GetChar(int index) const {
 if ((index >= 0) && (index <= str _ length))
  return str _ buf[index];
 else
  return 0;
}

void String :: AppendStr( const char *sub _ str) {
 char *temp _ str;
 str _ length = str _ length + strlen( sub _ str);
 temp _ str = new char[str _ length + 1]; // Allocate buffer
 strcpy (temp _ str, str _ buf);      // Copy old buffer
 strcat(temp _ str, sub _ str);      // Concatenate both strings
 delete [] str _ buf;
 str _ buf = temp _ str;
}

// Implementation of destructor
String :: ~String() {
 delete [] str _ buf;
```

continued on following page

Box 6. continued

```
}

//****************************
//     main()
//****************************
main() {

// Object of type String
 String string1( "Montana State University" );

// Operations using String class
 cout << "\nstring is: ";
 string1.Display();
 cout << "\nlength of string: " << string1.GetLength();
 cout << "\nfirst string character: "
 << string1.GetChar(0) << "\n";
// Appending a substring
 cout << "appending the sub-string: - Northern\n"
 << "string now is: ";
 string1.AppendStr( " - Northern\n\n");
 string1.Display();

 return 0;

}
```

ject signature, then the processing is automatically directed to the corresponding constructor, which can also preserve the object's type by storing an associated code. This action of the constructor is consistent with the notion of function overloading. Processing routines can dereference the object type and proceed accordingly.

Implementing the Object Classifier

The following program (Box 7) shows the processing for implementing a class named GeoFigure with four constructors: a default constructor that zeroes all the variables, and three parameterized constructors, one for each object signature. The constructors of the GeoFigure class perform object classification as the objects are created.

Observe in the program SAMP-07.CPP that processing operations that are often the client's burden are now handled by the class. The classifier class encapsulates knowledge about each object type, which is encoded and preserved with its signature. Thereafter, client code need not classify objects into squares, rectangles, or parallelograms, since this information is held in the class and handled transparently by its methods. The objects created by the classifier class know not only their dimensions but also their geometrical type, which in turn defines the processing operations necessary for performing calculations such as the area.

A classifier class is appropriate whenever there are objects with different signatures and their signatures determine the applicable processing operations or methods. Figure 13 is a generalized diagram for a classifier class; it can serve as a template for implementing this type of processing.

COMPOSING MIXED OBJECTS

Libraries, toolkits, and frameworks often provide two types of services to a client. A first level service performs the more elementary operations. A second-level service, called an aggregate or composite, allows combining several primitives to achieve a joint result. The pattern is referred to as a composite pattern. The composite is based on the notion of a Glyph, which is a class that encompasses all objects in

a document (Calder, 1990). Composite patterns are found in most object-oriented systems, including the original View class of Smalltalk (Krasner & Pope, 1988). Also in financial applications, aggregation of assets in a portfolio have been modeled using a Composite class (Birrer & Eggenschwiler, 1993).

Box 7.

```
//************************************************************
// C++ class template for an object classifier class
// Filename: SAMP-07.CPP
//************************************************************

#include <iostream.h>
#include <math.h>

//***************************
//    classes
//***************************
class GeoFigure {
private:
 float dim1;
 float dim2;
 float dim3;
 int fig _ type;
public:
// Declaration of four constructors for GeoFigure class
 GeoFigure();
 GeoFigure(float);
 GeoFigure(float, float);
 GeoFigure(float, float, float);
// Area() method uses object signature
 float Area();
```

continued on following page

Box 7. continued

```
};

// Parameterless constructor
GeoFigure::GeoFigure() {
 dim1 = 0;
 dim2 = 0;
 dim3 = 0;
 fig _ type = 0;
}
// Constructor with a single parameter
GeoFigure :: GeoFigure(float x){
 dim1 = x;
 fig _ type = 1;
}
// Constructor with two parameters
GeoFigure :: GeoFigure(float x, float y){
 dim1 = x;
 dim2 = y;
 fig _ type = 2;
}
// Constructor with three parameters
GeoFigure :: GeoFigure(float x, float y, float z){
 dim1 = x;
 dim2 = y;
 dim3 = z;
 fig _ type = 3;
}

float GeoFigure::Area() {
 switch (fig _ type) {
  case (0):
   return 0;
  case (1):
   return dim1 * dim1;
  case (2):
   return dim1 * dim2;
  case (3):
   return dim1 * (dim2 * sin(dim3));
  }
 return 0;
 }
```

continued on following page

Box 7. continued

```
//***************************
//     main()
//***************************
main() {
 GeoFigure fig0;        // Objects with different signatures
 GeoFigure fig1(12);
 GeoFigure fig2(12, 6);
 GeoFigure fig3(12, 6, 0.6);

// Calculating areas according to object signatures
 cout << "\nArea of fig1: " << fig1.Area();
 cout << "\nArea of fig2: " << fig2.Area();
 cout << "\nArea of fig3: " << fig3.Area();
 cout << "\nArea of fig0: " << fig0.Area() << "\n";

 return 0;
}
```

A Graphics Toolkit

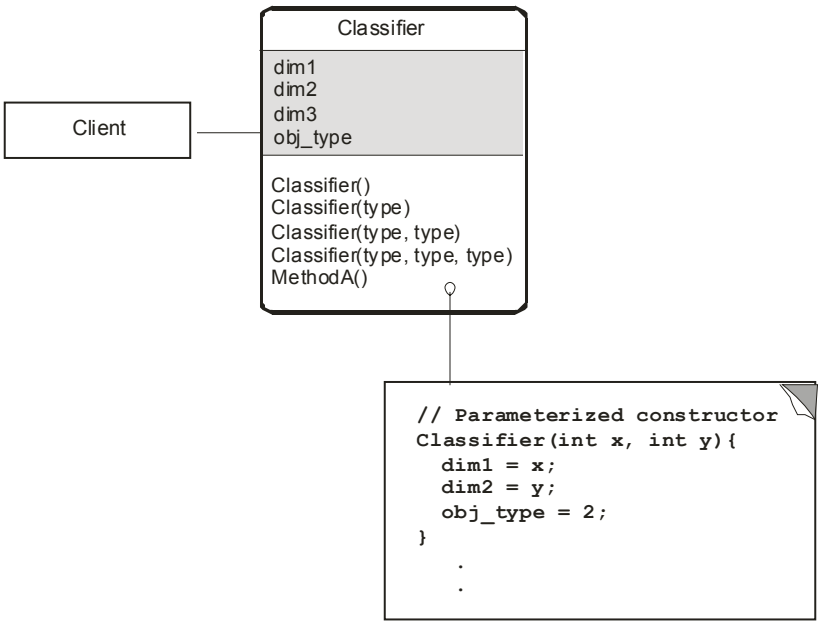
For example, a drawing program provides primitives for drawing geometrical figures such as lines, rectangles, and ellipses, for displaying bitmaps, and for showing text messages. A second-level function (the composite) allows combining of the primitive elements into a single unit that is handled as an individual program component. In the context of graphics programming, the term "descriptor" is often used to represent a drawing primitive and the term "segment" to represent a composite that contains one or more primitives.

Often a toolkit designer gives access to both the primitive and composite functions. In other words, the programmer using the drawing toolkit mentioned in the preceding paragraph would be able to create drawings that contained any combination of primitive and composite objects using a single, uniform, interface. In this example it may be useful to think of a composite as a list of instructions that includes the methods of one or more primitives. Figure 14 shows an image that contains both primitive and composite objects. The composite objects consist of a rectangle, a bitmap, an ellipse, and a text message. The primitive objects are text, a rectangle, and an ellipse.

The class structure for the sample toolkit could consist of a class for every one of the primitive operations and a composite class for combining several primitives into a drawing segment. An abstract class at the top of the hierarchy can serve to define an interface. Figure 15 shows the class structure for the graphics toolkit.

Note in Figure15 that the abstract class Image provides a general interface to the toolkit. The class Segment contains the implementation of the segment-level operations, the methods CreateSegment(), DeleteSegment(), and DrawSegment(). The drawing primitives are the leaves of the tree structure. Program SAMP-08.CPP (Box 8) is a partial implementation of the class diagram in Figure 15. Note that the classes Bitmap and Text were omitted in the sample code.

Figure 13. Object classifier class template



In the program SAMP-08.CPP the segment operation is based on an array of pointers. For this mechanism to work we need to implement run-time polymorphism since the composite (each instance of the Segment class) is created during program execution. In C++ run-time polymorphism can be achieved by inheritance and virtual functions. In this case the function Draw() is a pure virtual function in the abstract class Image, a virtual function in the class Segment, and is implemented in the leaf elements of the tree, which are the classes Line, Rectangle, and Ellipse. By making Draw() a simple virtual function we avoid making Segment into an abstract class. Therefore we can instantiate objects of Segment and still access the polymorphic methods in the leaf classes.

The actual code for implementing an array of pointers to objects has several interesting points. The array is defined in the private group of the Segment class, as follows:

```
Segment *ptr_ar[100];
```

This creates an array of pointers to the class Segment, and assigns up to 100 possible entries for each instantiation. The actual pointers are inserted in the array when the user selects one of the menu options offered during the execution of CreateSegment(). At this time a pointer to the Segment base class is reset to one of the derived classes, one of the Draw() methods at the leaves of the inheritance structure. The selected method is then placed in the pointer array named ptr_ar[]. For example, if the user selected the r (rectangle) menu option the following lines would instantiate and insert the pointer:

```
case ('r'):
   ptr_ss = &obj_rr; // Base pointer set
to derived object
   ptr_ar[n] = ptr_ss; // Pointer placed
in array
   n++; // Array index is bumped
   break;
```

Although it may appear that the same effect could be achieved by using a pointer to the method in the derived class, this is not the case. In this application a pointer to a derived class will be unstable and unreliable.

Figure 14. Primitive and composite objects in a graphics toolkit

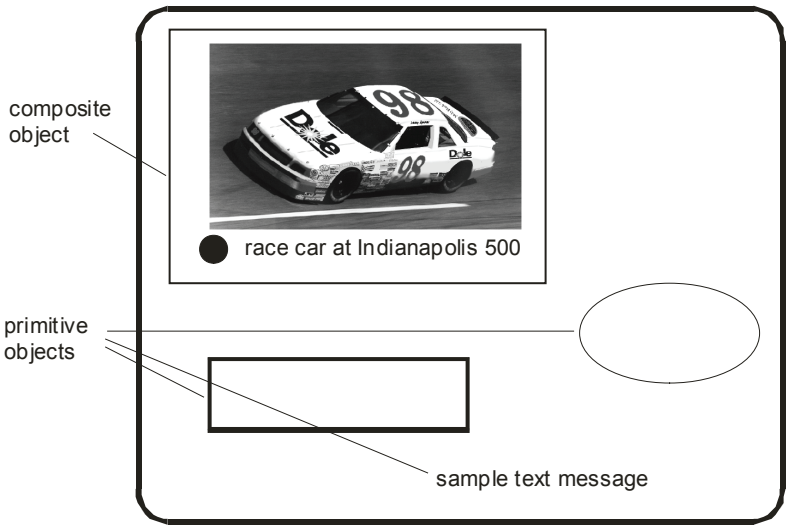
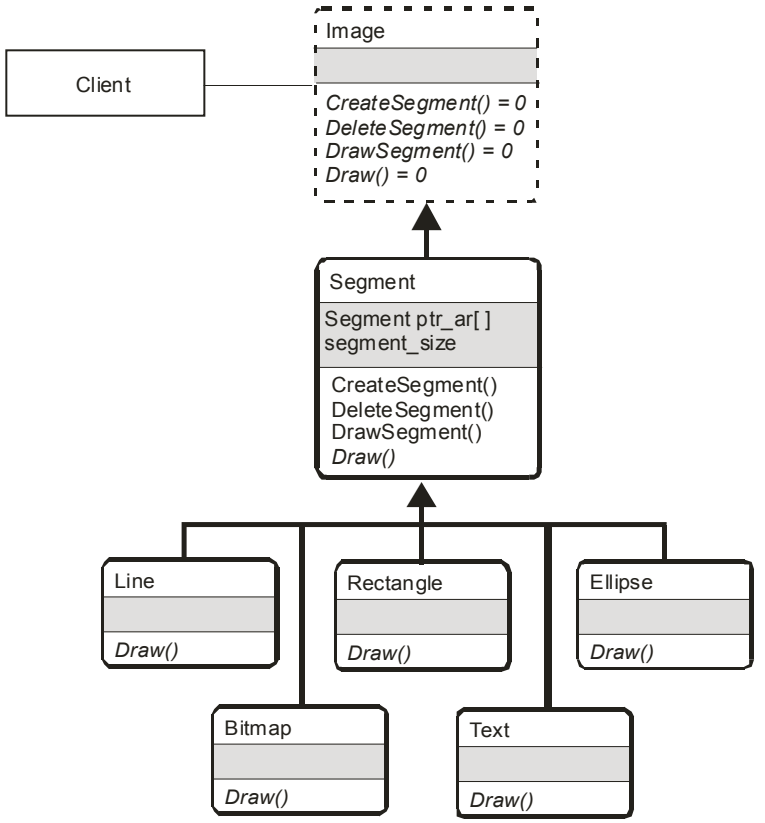


Figure 15. Tree structure for creating simple and composite objects



Box 8.

```
//************************************************************
// C++ program to illustrate the creation of simple and
// composite objects
// Filename: SAMP-08.CPP
//************************************************************

#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

//***************************
//    classes
//***************************
// Image class provides a general interface
class Image {
public:
 void virtual CreateSegment() = 0;
 void virtual DrawSegment() = 0;
 void virtual Draw() = 0;
};

class Segment : public Image {
private:
 Segment *ptr _ ar[100];       // Array of pointers
 int seg _ size;            // Entries in array
public:
 void CreateSegment();
 void DrawSegment();
 void DeleteSegment();
 void virtual Draw() { return; };
};


class Rectangle : public Segment {
public:
 void Draw() { cout << "\ndrawing a rectangle"; }
};

class Ellipse : public Segment {
public:
 void Draw() { cout << "\ndrawing an ellipse"; }
```

continued on following page

Box 8. continued

```
};

class Line : public Segment {
public:
 void Draw() { cout << "\ndrawing a line"; }
};

// Implementation of methods in Segment class
void Segment::CreateSegment() {
 char select;
 int n = 0;         // Entries in the array
// Objects and pointers
 Segment obj_ss;
 Segment *ptr_ss;
 Line   obj_ll;
 Rectangle obj_rr;
 Ellipse obj_ee;

 cout << "\n opening a segment...\n";
  cout << "Select primitive or end segment: "
  << "\n l = line"
  << "\n r = rectangle"
  << "\n e = ellipse"
  << "\n x = end segment"
  << "\n SELECT: ";
 do {
  select = getche();
  switch(select) {
  case ('l'):
   ptr_ss = &obj_ll;
   ptr_ar[n] = ptr_ss;
   n++;
   break;
  case ('r'):
   ptr_ss = &obj_rr;
   ptr_ar[n] = ptr_ss;
   n++;
   break;
  case ('e'):
   ptr_ss = &obj_ee;
   ptr_ar[n] = ptr_ss;
   n++;
```

continued on following page

Box 8. continued

```
    break;
   case ('x'):
    break;
   default:
    cout << "\nInvalid selection - program terminated\n";
    exit(0);
  }
  }
  while( select != 'x');
   seg _ size = n;
   cout << "\n closing a segment...";
}

void Segment::DrawSegment() {
 cout << "\n  displaying a segment...";
 for(int x = 0; x < seg _ size; x++)
  ptr _ ar[x]->Draw();
 cout << "\n  end of segment display ...";
 return;
}
//***************************
//     main()
//***************************
main() {
 Segment  obj _ s;
 Line   obj _ l;
 Rectangle obj _ r;
 Ellipse  obj _ e;

// Creating and drawing a segment
 cout << "\n\nCalling CreateSegment() method";
 obj _ s.CreateSegment();
 obj _ s.DrawSegment();
// Drawing individual objects
 obj _ l.Draw();
 obj _ r.Draw();
 obj _ e.Draw();

 return 0;
}
```

At the conclusion of the CreateSegment() method it is necessary to preserve with each object a count of the number of points that it contains. The seg_size variable is initialized to the number of pointers in the array in the statement:

```
seg _ size = n;
```

At the conclusion of the CreateSegment() method, the array of pointers has been created and initialized for each object of the Segment class, and the number of pointers is preserved in the variable seg_size. Executing the segment is a matter of recovering each of the pointers in a loop and executing the corresponding methods in the conventional manner. The following loop shows the implementation:

```
for(int x = 0; x < seg _ size; x++)
  ptr _ ar[x]->Draw();
```

PATTERN FOR AN OBJECT FACTORY

By eliminating all the unnecessary elements in the class structure of Figure 15 we can construct a general class pattern for creating simple and composite objects. This version of the Composite class can be considered as a simple object factory which uses an array to store one or more pointers to objects. In addition, each object of the Composite class keeps count of the number of pointers in the array. This count is used in dereferencing the pointer array.

Alternatively, the pointer array can be implemented without keeping a pointer counter by inserting a NULL pointer to mark the end of the array. This NULL pointer then serves as a marker during dereferencing. In either case, the corresponding methods in the leaf classes are accessed by means of the pointers in the array. Method selection must be implemented by dynamic binding. In C++ the polymorphic method must be virtual in the composite class. The pattern is shown in Figure 16.

A Simplified Implementation

We can simplify the concept of primitive and composite objects, as well as their implementation in code, by allowing a composite that consists of a single primitive object. For example, if in Figure 15 we permit a segment that consists of a single primitive, then the client needs never to access the primitives directly. This makes the interface less complicated. In many cases this option should be examined at design time.

RECURSIVE COMPOSITION

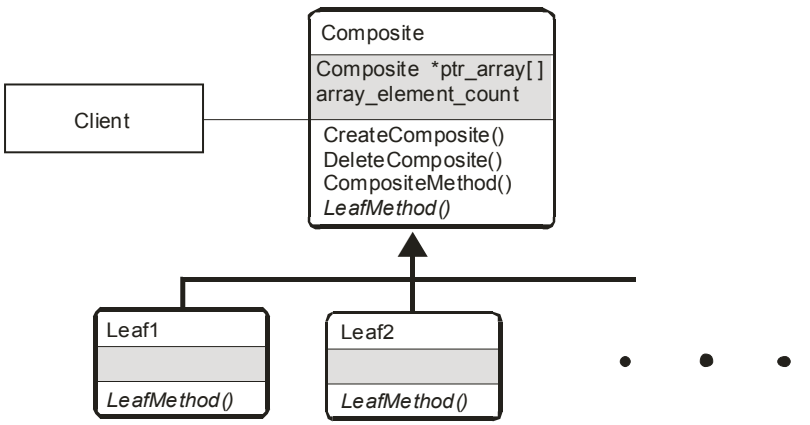
In previous sections we considered the case of a composite class that contains simple or composite objects. We also looked at the alternative of a composite object that consists of a single primitive as a way of simplifying the interface. However, we have not yet considered the possibility of a composite object containing another composite (Gamma et al., 1995). Based on the class structure shown in Figure 15 we can construct an object diagram in which a Segment can contain another Segment as shown in Figure 17. Note that in this case we have preserved the distinction between primitives and composites.

Implementation Considerations

In the previous example recursive composition is based on a nested reference to the CreateSegment() method of the Segment class. However, recursion is often accompanied by a new set of problems; this case is no exception. The first consideration is to access the CreateSegment() method. Three possibilities are immediately evident:

1. Since CreateSegment() is called from within the class, it can be referenced without instantiating a specific object.
2. We can access the CreateSegment() method by means of a this pointer. In fact, this is a

Figure 16. Pattern for an object factory class



different syntax but has the same result as the previous case. In both instances the current object is used.

3. We can create a new object and use it to access the CreateSegment() method.

Which method is suitable depends on the problem to be solved. Figure 18 shows a class diagram for recursively accessing the CreateSegment() method using the original object.

The program SAMP-09.CPP (Box 9) shows the implementation of the class diagram in Figure 18.

Several points in the code merit comment. In the first place notice that recursion occurs on the same object originally referenced at call time. This is accomplished by means of the C++ this pointer, in the following statement:

```
this->CreateSegment();
```

The CreateSegment() method could have been accessed without the this pointer since it is allowed, within the same class, to access methods directly. When CreateSegment() is accessed recursively, all the local variables are automatically reinitialized. Since the program requires a count of the number of pointers in the pointer array, we made the iteration counter (variable n) a global variable

Figure 17. Object diagram for recursive composition

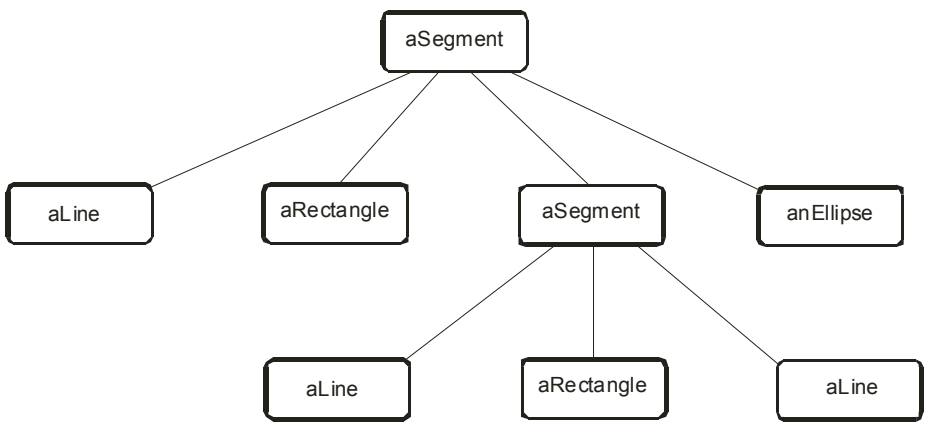
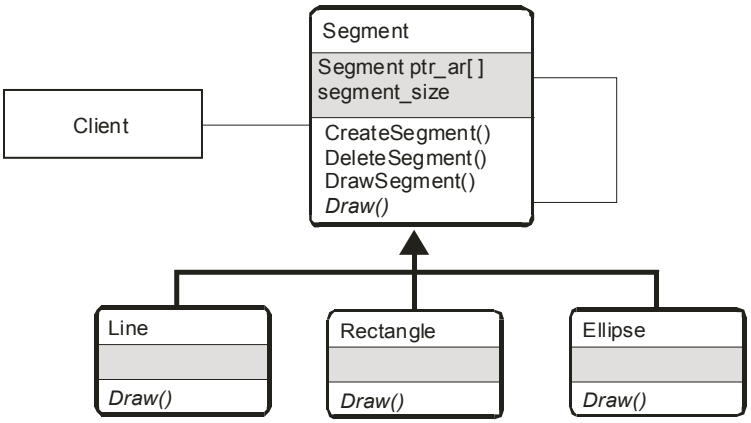


Figure 18. Class diagram for recursive composition



and created a switch variable named instance. This variable is set to 1 when CreateSegment() is called recursively, determining that counter variable n is not cleared on entry to the method. The result is that n holds the number of pointers inserted in the pointer array, whether they were entered directly or recursively.

A Recursion Pattern

The pattern for recursive composition is similar to the one in Figure 16, except that in recursion there is an arrow pointing to the same composite class. This is shown in Figure 19.

CONCLUSION

Software design is one of the most laborious and time-consuming phases of program development. In object-oriented systems design reuse is based on classes and object structures that solve a particular design problem and on the assumption that these class structures can be applied to other similar problems. The most recent approach to design reuse is based on class associations and relationships called patterns or object models. In this sense a programming problem is at the origin of every pattern.

We have described the use of design patterns and template classes as reusable components in program design. The discussion has been complemented with class diagrams and examples implemented in code. The chapter introduces the notion of a class template as a structure that describes functionality and object relations within a single class, while patterns refer to structures of communicating and interacting classes.

REFERENCES

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A pattern language.* New York: Oxford University Press.

Beck, K., & Johnson, R. (1994, July). *Patterns generate architectures*. Paper presented at the European Conference on Object-Oriented Programming, Bologna, Italy (pp. 139-149). Springer-Verlag.

Birrer, A., & Eggenschwiler, T. (1993, July). *Frameworks in the financial engineering domain: An experience report*. Paper presented at the European Conference on Object-Oriented Programming, Kaiserslautern, Germany (pp. 21-35). Springer-Verlag.

Box 9.

```
//************************************************************
// C++ program to illustrate recursive composition
// Filename: SAMP-09.CPP
//************************************************************

#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

//***************************
//    classes
//***************************
class Segment {
private:
 Segment *ptr _ ar[100];
 int seg _ size;            // Entries in array
public:
 void CreateSegment();
 void DrawSegment();
 void DeleteSegment();
 void virtual Draw() { return; };
};

class Rectangle : public Segment {
public:
 void Draw() { cout << "\ndrawing a rectangle"; }
};

class Ellipse : public Segment {
public:
 void Draw() { cout << "\ndrawing an ellipse"; }
};

class Line : public Segment {
public:
 void Draw() { cout << "\ndrawing a line"; }
};
// Global variable for controlling recursive implementation
// of the CreateSegment() method
 int n;
 int instance = 0;
```

continued on following page

Box 9. continued

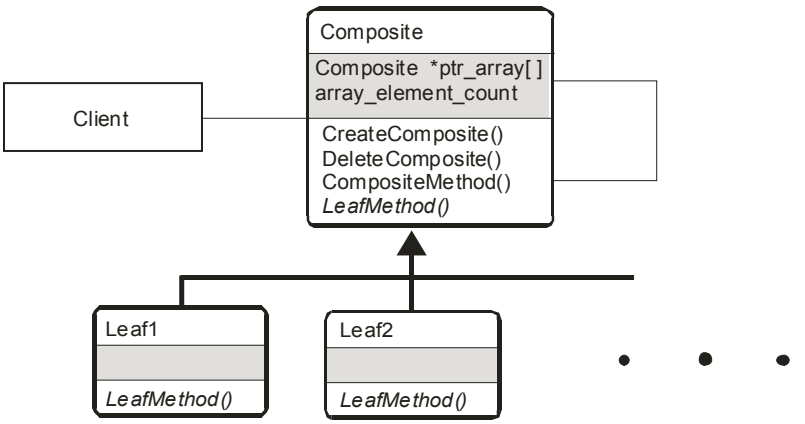
```
// Implementation of methods in Segment class
void Segment::CreateSegment() {
 char select;
// Entries in the array
// Objects and pointers
 Segment obj _ ss;     // An object
 Segment *ptr _ ss;     // Pointer to object
 Line   obj _ ll;     // Object list
 Rectangle obj _ rr;
 Ellipse obj _ ee;

 if(instance == 0)
  n = 0;

  cout << "\n opening a segment...\n";
  cout << "Select primitive or end segment: "
 << "\n l = line"
 << "\n r = rectangle"
 << "\n e = ellipse"
 << "\n n = nested segment"
 << "\n x = end segment"
 << "\n SELECT: ";
 do {
  select = getche();
  switch(select) {
  case ('l'):
   ptr _ ss = &obj _ ll;     // Pointer to object initialized
   ptr _ ar[n] = ptr _ ss;     // and stored in array
   n++;
   break;
  case ('r'):
   ptr _ ss = &obj _ rr;
   ptr _ ar[n] = ptr _ ss;
   n++;
   break;
  case ('e'):
   ptr _ ss = &obj _ ee;
   ptr _ ar[n] = ptr _ ss;
   n++;
   break;
  case ('n'):
```

continued on following page

Box 9. continued

```
    cout << "\n   nested segment...";
    instance = 1;
    this->CreateSegment();
    cout << "\n   nested segment closed ...";
    cout << "\n SELECT: ";
    continue;

   case ('x'):
    break;
   default:
    cout << "\nInvalid selection - program terminated\n";
    exit(0);
  }
  }
  while( select != 'x');
   seg _ size = n;
   cout << "\n closing a segment...";
   instance = 0;          // Reset instance control
}

void Segment::DrawSegment() {
 cout << "\n  displaying a segment...";
 for(int x = 0; x < seg _ size; x++)
  ptr _ ar[x]->Draw();
 cout << "\n  end of segment display ...";
 return;
}
//***************************
//     main()
//***************************
main() {
 Segment  obj _ s;

// Creating and drawing a segment with possible nested
// segments
 cout << "\n\nCalling CreateSegment() method";
 obj _ s.CreateSegment();
 obj _ s.DrawSegment();

 return 0;
}
```

Figure 19. Pattern for recursive composition



Calder, P. R., & Linton, M. A. (1992, October). *The object-oriented implementation of a document editor*. Paper presented at the Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, Vancouver, British Columbia, Canada (pp. 1-15). ACM Press.

Coplien, J. O. (1992). *Advanced C++ programming styles and idioms*. Reading, MA: Addison-Wesley.

Deutsch, L. P. (1989). Design reuse and frameworks in the Smalltalk-80 system. *Software reusability, Volume II: Applications and experience* (pp 57-71). Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Krasner, G. E., & Pope, S. T. (1988, August/September). A cookbook for using the model-view controller user interface paradigm in Smalltalk 80. *Journal of Object-Oriented Programming, 1*(3)26-49.

Vlissides, J. M., & Linton, M. A. (1990, July). Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems, 8*(3), 237-269.

KEY TERMS

Concealer Pattern: A class pattern designed to hide an inheritance structure so that its existence and operation becomes transparent to client code.

Chain Reaction Pattern: A class pattern that focuses on the property of classes that instantiate objects of other classes higher in the class hierarchy, thus starting a chain reaction in the production of objects.

Class Template: A structure of interactive objects that implements a particular functionality within a single class.

Design Pattern: A structure of interactive classes and communicating objects that provide a solution to a software design problem.

Interface Pattern: A pattern that provides access to other classes that implement a desired functionality.

Mixer Pattern: A class that inherits from two or more classes, thus providing a unified interface to the methods and objects in these classes.

Object Chaining: A chain of objects that successively examines an operand pass to them and makes some decision based on the result of this examination.

Object Classifier: A template for a class with constructors that create objects with different numbers and types of parameters.

Object Factory: A composite class that creates both simple and composite objects.

Unifier Pattern: A class pattern designed to present a unified and friendly interface to a set of classes that perform different operations.