SKILL

EIGHTEEN

DEVELOPING MORE ADVANCED APPLICATIONS

- Completing a short project—an improved point-of-sale terminal
- Choosing a programmer-user interface
- Choosing an end-user interface
- Choosing and implementing classes
- Completing another short project—a satellite simulation

The ability to develop applications is the essence of the programmer's profession. This ability has been continuously developed in most Skills in this book. In Skill 18, you will continue to improve it by completing two short projects:

- Developing an improved point-of-sale terminal using classes
- Developing a satellite simulation using classes

Short Project 1—an Improved Point-of-Sale Terminal

You will now learn how to transform the point-of-sale terminal you developed in Skill 9 into an object. The goal is to use as many of the features as possible that were discussed in Skill 9 (using the `c3sale2.cpp` program).

The functionality of the terminal will be the same as the previous terminal's functionality. All I want to do now is to show you how to transform the terminal into an object. This will allow you to work on interesting expansions in the next Skills. You may benefit from looking back to Skill 9, and from running the `c3sale2.cpp` program once more.

Choosing the Programmer-User Interface

As you progress in your programming career, more people will use the programs that you develop. Some people may use your software as end users. For example, a clerk at a store using your terminal program is an end user. This person does not know how your program was built. All he or she cares to know is how to hit the right keys to do his or her job. However, there is another kind of user of whom you must be aware.

The Programmer User

In a world in which software is extremely complex, programmers use programs that were developed by other programmers to produce better software. You have been using the programs in `franca.h`, which I developed. You are a programmer user of my software. However, you use this software to help you produce your own software, which may have an end user who knows nothing about programming.

When you design software for programmer users, you must also exercise care to make it simple to use.

The current project consists of creating a class of point-of-sale terminals, so an object of this class can be used in a program to generate a final product—the terminal.

In this case, you will implement the class, and you will be the one to use it. You will be your own programmer user. However, this may not always be the case!

Interfacing

There may be many ways to implement the terminal as a class. One possibility is to have a terminal that is only created and told to operate. This means that the main program, besides creating the terminal, only has to issue a call to one function to put it to work.

The main function could look as follows:

```
void mainprog()
{
  terminal cashregister;
  cashregister.operate();
}
```

This will indeed be a very simple interface for the programmer who uses your class.

A class that does everything by itself, like this one, may not be the best idea. You may consider including some operations that allow the programmer to manipulate the object with greater flexibility. For example, you could provide member functions that allow the programmer to write to the boxes or to process one customer at a time.

Choosing the End-User Interface

The end-user interface will be the same one used in `c3sale2.cpp`. It is a good idea to keep the user interface as long as the customers are happy with it. Anytime you change the user interface—move a box from one place to another, ask a question in a different way, etc.—the user will spend some time to get used to the changes. Sometimes the user will not like the changes!

Declaring and Defining Classes

The `terminal` class should contain most of the variables and objects that were present in the previous program. This could lead to the following declaration:

```
class terminal
{
 protected:
  float tax,amount,price,saletotal,change;
  Box Price,Cur_Total;
  Box Saletotal,Amount,Change;
  void items(); // Process items in a sale
 public:
  terminal();
  void operate();
};
```

Notice that in the `c3sale2.cpp` program, you could initialize the boxes. In this case, you cannot initialize the boxes in the class declaration. This task has to be left to the constructor.

No initial values can be set in a class declaration.

The Constructor Code

The constructor code is relatively simple. The only reason it looks a little long is that the boxes have to be positioned and provided with a label.

```
terminal::terminal()
{
  tax=ask("Enter the sale tax %")/100.;
  Price.place(450,100);
  Price.label("Price:");
  Cur_Total.place(450,140);
  Cur_Total.label("Subtotal:");
  Saletotal.place(450,180);
  Saletotal.label("Total Sale:");
  Amount.place(450,220);
  Amount.label("Amount Tendered");
  Change.place(450,260);
  Change.label("Change:");
}
```

Figure 18.1 shows these boxes after processing a sale.

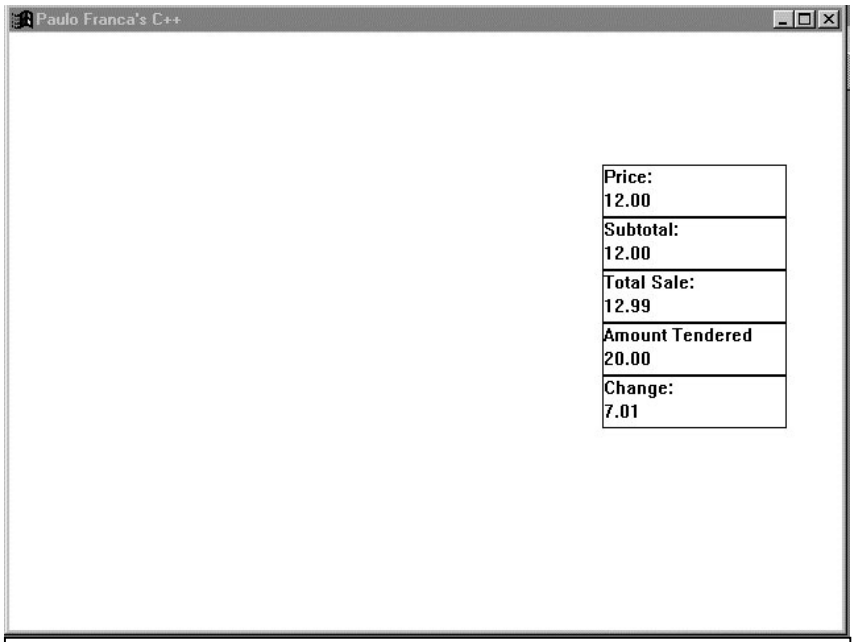

Fig 18.1 Boxes after sale processing

The *operate()* Member Function

The `operate()` member function is the function that actually makes the terminal work. You may notice that this function closely resembles what was done in the previous program. A loop handles the sale to each customer. After each sale, the total is displayed and the change is computed.

To simplify the code, the loop that handles each item in a sale is handled by another function `items()`. Still, this is just like the `c3sale2.cpp` program. However, notice that the programmer user is not supposed to invoke the `items()` member function. This function was designed to be used internally only. For this reason, this member function is not public.

Here is the code for the `operate()` function:

```
void terminal::operate()
{
   float tendered,change;
   for(;;)
   {
     saletotal=0.;
     Price.say(" ");
     Cur_Total.say(" ");
     Saletotal.say(" ");
     Amount.say(" ");
     Change.say(" ");
     items();
     Saletotal.say(saletotal);
     amount=ask("Enter amount tendered:");
     Amount.say(amount);
     change=amount-saletotal;
     Change.say(change);
     if(!yesno("Another customer?"))break;
   }
}
```

Notice that all the values and the boxes displaying them are cleared at the beginning of each sale. Then, the `items()` member function handles all the items in the sale. After doing this, all the boxes are updated with the pertinent information.

The program above uses two very similar names for different things: there is a box named `Change` (capitalized) and a variable named `change` (lowercased).

The *items()* Function

The `items()` function serves a purpose similar to that of the `getitems()` function used in `c3sale2.cpp`. It adds all the items purchased by the same customer.

The code for the `items()` function is as follows:

```
void terminal::items()
{
  for(;;)
  {
    price=ask("enter the price:");
    saletotal=saletotal+price;
    Price.say(price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
  saletotal=saletotal*(1+tax);
  Saletotal.say(saletotal);
}
```

IMPROVEMENTS

The most relevant improvement to this terminal would be to replace input from a price with input from a code. This is what most present-day point-of-sale terminals do. Currently, you don't have the means to make this improvement. Further versions of this project will be presented in upcoming Skills.

The Complete Program Listings

The complete code for this project can be found in `c6term.h` and `c6term.cpp`.

The Main Program *c6term.cpp*

Here is the code for `c6term.cpp`:

```
#include "franca.h"
#include "c6term.h"
void mainprog()
{
  terminal cashregister;
  cashregister.operate();
}
```

The Header File *c6term.h*

Here is the code for `c6term.h`:

```
#ifndef C6TERM_H
#define C6TERM_H
#include "franca.h"
class terminal
{
 protected:
  float tax,amount,price,saletotal,change;
  Box Price,Cur_Total;
  Box Saletotal,Amount,Change;
  void items(); // Process items in a sale
 public:
  terminal();
  void operate();
};
terminal::terminal()
{
  tax=ask("Enter the sale tax %")/100.;
  Price.place(450,100);
  Price.label("Price:");
  Cur_Total.place(450,140);
  Cur_Total.label("Subtotal:");
  Saletotal.place(450,180);
  Saletotal.label("Total Sale:");
  Amount.place(450,220);
  Amount.label("Amount Tendered");
  Change.place(450,260);
  Change.label("Change:");
}
```

```
void terminal::items()
{
  for(;;)
  {
    price=ask("enter the price:");
    saletotal=saletotal+price;
    Price.say(price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
  saletotal=saletotal*(1+tax);
  Saletotal.say(saletotal);
}

void terminal::operate()
{
   float tendered,change;
   for(;;)
   {
     saletotal=0.;
     Price.say(" ");
     Cur_Total.say(" ");
     Saletotal.say(" ");
     Amount.say(" ");
     Change.say(" ");
     items();
     Saletotal.say(saletotal);
     amount=ask("Enter amount tendered:");
     Amount.say(amount);
     change=amount-saletotal;
     Change.say(change);
     if(!yesno("Another customer?"))break;
   }
}
#endif
```

Short Project 2—Satellites

The program `c5stars.cpp`, developed as a project in Skill 15, will now be reviewed and improved with the use of objects. Indeed, the circles were used to represent a star (the Sun), a planet (Earth), and a satellite (the Moon).

Generally speaking, these objects orbit other objects at a particular distance and angular velocity. If you think abstractly instead of using the usual nomenclature that differentiates between stars, planets, and satellites, you may realize that all of them can be treated like satellites.

Indeed, Earth, as well as the other planets in our solar system, is a satellite orbiting the Sun. You can consider that the Sun itself is orbiting any distant body at speed 0, or that it is orbiting itself. This abstract thinking leads to the idea of creating a new class of objects—satellites.

Satellites can be used to simplify the `c5stars.cpp` program, and can also be used in other applications.

A satellite can be represented by a circle that moves in a circular pattern around a specific point or, to be more interesting, around another satellite (including itself). Besides all the usual features of a circle, satellites have the following features:

- A planet to orbit. This planet gives the coordinate of the point they are orbiting.
- A distance from that planet. This is the radius of the orbit.
- An angular velocity for the movement.

- A current angle in the orbit. This angle and the radius constitute the polar coordinates of the satellite, relative to the planet.

In addition, satellites should know how to move from one place to the next place as time passes. However, instead of using a time-dependent equation to compute the position, you can define a function `move` that will be invoked for each drawing frame (every 30th of a second). This function computes the coordinates of the satellite after each frame time elapses.

Designing the Satellite Class

Here is the declaration of the `satellite` class:

```
class satellite: public Circle
{
  protected:
   float xplanet,yplanet;
   float radius,omega,wspeed;
  public:
   satellite();
   move();
};
```

Data members include the planet's coordinates (notice that you don't need the planet—you only need to know where it is), the orbit's radius (`radius`), the current angle (`omega`), and the angular velocity per unit of time (`wspeed`).

There are a few design choices concerning the angular velocity. Ultimately, you need to know how many radians the satellite moves per time frame (a 30th of a second). However, you may choose to allow a user of your class (including yourself) to specify it as degrees per second, even though you convert this value and store it as radians per time frame.

Using a Constructor

A constructor function is always desirable, so we can color, size, place, and perform other initialization tasks with the object. The `move` function, discussed in an earlier Skill, should be called to move the satellite to a new location in the orbit after the time frame has elapsed.

It is clear that you could use a function such as `move(t)` to determine the current time and to compute the position at this time.

Unfortunately, the class declaration above is not appropriate, yet. How will we set values for the data members?

One alternative is to allow the constructor to take these values as parameters. This would work, but once a value is set, it cannot be changed unless one of the following items is true:

- The data member is public.
- There is a member function that allows you to change the values.

Even though you may think that some of the data members, such as the radius, the coordinates of the planet, or the angular velocity, will never change, you may be amazed by what you can accomplish with a little flexibility.

Member functions can be included to change the following items:

- The coordinates of the planet
- The radius
- The angular velocity

Specifying a Coordinate System for the Satellites

Finally, as you may have noticed in `c5stars.cpp`, it will come in very handy to convert from polar coordinates to cartesian coordinates. Include a `polarxy()` function in the class. Here is the declaration proposed for this design:

```
class satellite: public Circle
{
  protected:
   float xplanet,yplanet;
   float radius,omega,wspeed;
   void polarxy(float r,float theta,float x0,
                float y0,float &x,float &y);
  public:
   satellite();
   void center(float x,float y);
   void dist(float rad);
   void angle(float ang);
   void speed(float ws);
   void move();
   void center(satellite &another);
};
```

There are two `center()` functions that allow you to change the planet's coordinates. These functions allow you to specify that a planet revolves around a given point (*x*,*y*) or around another planet.

In this case, the `polarxy()` function can be modified so no arguments are needed. This may be a good idea, but it is left unmodified, so we reuse the same code.

Implementing the Class

Once the class declaration is ready, the code for the functions can be developed.

The Constructor Code

Here is the constructor code:

```
satellite::satellite()
{
   color(2);
   xplanet=yplanet=0;
   radius=0;
   omega=wspeed=0;
}
```

There is nothing special about this constructor. Notice that the satellite's color can be modified at any point, since the `color()` function is public for the circles.

Changing the Center

The functions to change the planet's coordinates are as follows:

```
void satellite::center(float xc,float yc)
{
   xplanet=xc;
   yplanet=yc;
}
```

```
void satellite::center(satellite &another)
{
   center(another.x,another.y);
}
```

Do you see anything interesting in this code? Why does the second function use the *x* and *y* coordinates of the other planet? Aren't these coordinates protected?

Indeed, they are! However, protected members can be used by member functions of the same or derived class. Since the second function is a member function for `satellite`, which is derived from `Circle`, it is possible to access the coordinates. Not only the coordinates of this satellite are accessible, but the coordinates of any other satellite that is used inside the member function are accessible.

It makes the use of satellites much more convenient to include the second version of the `center()` function. It is definitely easier to say

```
Moon.center(Earth);
```

than it is to determine Earth's coordinates to include in the function call.

Updating the Radius, Angle, and Angular Velocity

The functions to update the radius, the current angle, and the angular velocity are simple.

```
void satellite::dist(float rad)
{
   radius=rad;
}
void satellite::angle(float ang)
{
   omega=ang;
}
void satellite::speed(float ws)
{
   wspeed=ws;
}
```

Moving Around

Finally, the function to move the satellite from one position to the next is as follows:

```
void satellite::move()
{
   omega=omega+wspeed;
   polarxy(radius,omega,xplanet,yplanet,x,y);
}
```

It is very simple, isn't it? All you do is to increment the current angle, and then compute the new cartesian coordinates. There is no need to invoke the `place(x,y)` function. The cartesian coordinates are directly updated with this function call.

The New Stars Program

Once the satellite class is available (`c6satell.h`), you can write a new version of the `c5stars.cpp` program (`c6sat.cpp`).

You can declare objects and initialize them as follows:

```
const float pi2=2*3.14159;
Stage universe;
Clock timer,sidereal;
satellite Sun,Earth,Moon;
Sun.resize(40);
Sun.origin(320,240);
Sun.scale(1.,-1.);
Earth.dist(120);
Earth.speed(pi2/36.5/30.);// In a 30th of a second
Earth.center(Sun);
Moon.resize(12);
Moon.dist(80);
Moon.color(3);
Earth.color(4);
Moon.speed(pi2/2.8/30.);  // In a 30th of a second
universe.insert(Earth);
universe.insert(Moon);
Sun.show();
```

The animation loop is simple.

```
for(;sidereal.time()<36.5;)
  {
    Earth.move();
    Moon.center(Earth);
    Moon.move();
    universe.show();
    timer.watch(.033);
    timer.reset();
    universe.erase();
  }
```

Why is the Moon told to center using Earth inside the loop? Couldn't this be done only once before the loop, as it was done with the Sun? No, it could not!

The problem is that Earth's coordinates are changing, and what we actually give the Moon are the new coordinates of Earth. This procedure is not needed for the Sun, because it does not move. Now do you see why it was useful to be able to change the center coordinates?

From Outer Space to Inner Space

What else can you do with satellites? Electrons are also satellites of the nucleus of an atom.

You can write a program to simulate electrons orbiting a nucleus just as easily as you wrote the previous program. If you want 10 electrons orbiting the nucleus, declare 10 satellites, initialize them, move them.… Wow—it may become an extensive program! Well, not really—wait until you learn about arrays!

Are You Experienced?

Now you can…

Choose appropriate classes to implement your applications

Evaluate possible improvements to your designs