The Power of Object-Oriented Frameworks

Part 1: Introduction to Framework Technology

Dr. Fons Rademakers

(Fons.Rademakers@cern.ch)
CERN & Hewlett-Packard
1211 Geneva 23, Switzerland

To be published in ‘‘Interface Magazine’’ (http://www2.cqpub.co.jp/~inter/)

Introduction

In High Energy Physics (HEP) we try to find the elementary building blocks of the universe and the forces that work on them. To search for these elementary particles we need to recreate an environment that existed 10^{-43} seconds after the Big Bang. Using giant accelerators, like the Large Electron Positron (LEP) collider, which has a circumference of 27 km, we can recreate this environment. To detect the particles produced by the accelerators we use very large and complex detectors. These detectors have typically a size of 10 x 10 x 10 meters and are constructed and operated by teams of several hundreds of physicists. Currently we are constructing at CERN (Geneva, Switzerland) a new accelerator, the Large Hadron Collider (LHC), which will allow us to probe even deeper into the beginning of the universe. The LHC uses the same 27 km tunnel as LEP and is scheduled to start operating in 2005. There will be four main detectors measuring the results of the collisions. Each of these four detectors will record data with a rate of about 100 MByte/s which results in about 1 PByte (Peta=10^{15}) of data per year. The LHC is expected to stay in operation till at least 2020.

Computing in HEP

Traditionally all programs in HEP were written in FORTRAN. To correct the omissions of the FORTRAN language, clever and powerful data management subroutine libraries were written that allowed us to build complex programs that could handle the formidable data processing tasks. Many of these 20 year old data management subroutine libraries are still in use today due to the very long life time of the HEP experiments.

During the years we were using FORTRAN, computer science was not standing still. New languages and programming paradigms were developed and maturing in the world around us. Progress in these areas has been substantial and we can no longer afford to ignore them. Therefore, the people building the experiments that will run in the LHC era decided that now is the right time to switch to Object-Oriented Programming (OOP) techniques and in particular to adopt C++ as programming language.

Although a majority of physicists agree that OOP and C++ are a sensible choice, their actual use in the scientific community is still very limited. One of the main obstacles has been the lack of a suitable OO framework for scientific data analysis. Such a framework should provide, among others, basic database functionality, histogramming, statistical analysis and visualization tools, container classes, a powerful macro language, GUI and OS interfaces and platform independence.

In the next sections I will give an overview of the main benefits of Object-Oriented Programming and of Object-Oriented Frameworks in particular.

Object-Oriented Programming

The great programming paradigm of the past four decades has been procedure-orientation. This paradigm has served us well and was supported by procedural programming languages like FORTRAN, C, Cobol, Pascal, etc. About two decades ago object-orientation was introduced as new powerful paradigm. It was supported by languages like Simula, and later, Smalltalk. These languages were esoteric and not very performant. Only with the recent introduction of C++, which inherited a large part of its syntax from the very popular C language and which does not sacrifice any efficiency compared to C, did object-orientation become a viable option for large programming projects.

Object-Oriented design and programming is based on the fundamental ideas of objects, classes, methods, messages, encapsulation, inheritance, polymorphism and dynamic binding. Some of these are new words for old ideas, while others are names for new concepts in software design.

The OOP idea of a module is different from the procedural idea of a module. In the OOP paradigm, the purpose of a module is to encapsulate data along with processing functions, making the function/data cluster one simple, compact, elegant unit of software. Such modules are called objects and in their purest form objects contain functions called *methods* and states called *instance variables* (in C++ terminology: *member functions* and *data members*, respectively).

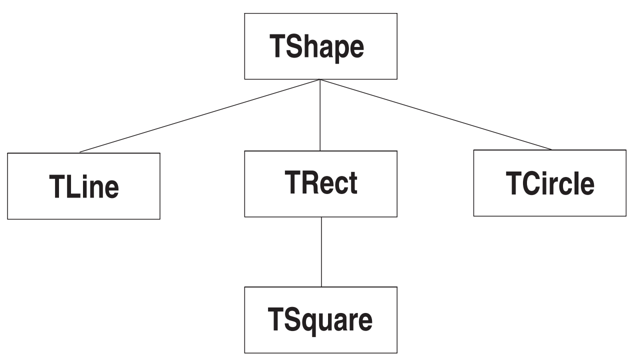

Figure 1 The shape class hierarchy

All objects are instances of classes. A *class* is like a blueprint for the creation of objects. It describes the data members and methods an object will have. Each object will have its own private copy of the data members, while the methods are shared by all objects of a class. Using *inheritance*, classes can be specialized. For example, a geometrical shape can be described by a class called `TShape`[1]. This `TShape` class will contain, among others, the *x* and *y* positions of the origin of the shape, since all shapes will have an origin. Also the `TShape` class will describe the basic protocol (i.e. the member functions) that all shape objects should be able to respond to, like `Draw()`. Specializations of the `TShape` class typically are shapes like lines, `TLine`, rectangles, `TRect`, circles, `TCircle`, etc. Each of these specialized classes will inherit the origin and the `Draw()` protocol from the `TShape` class, while adding their specific data members and `Draw()` method implementation. The `TShape` class is called a *base* class and the specializations are called *derived* classes. In the above example the `TShape` class actually is an *abstract* base class, since a `TShape` is not a well defined geometrical object that can exist by itself (how to draw a `TShape`?). Abstract classes describe common data members and methods, but one can not instantiate objects of abstract classes. Using inheritance one can build *class hierarchies*, see Figure 1. Inheritance describes so called ‘‘is a’’ relationships. For example a `TLine` ‘‘is a’’ `TShape`. After you’ve instantiated an object, its inherited behaviour is determined by its place in the class hierarchy, but it may also be part of a *containment hierarchy*: that is, it can contain and be contained by other objects. Containment relationships are also called ‘‘has a’’ relationships. For example, a canvas that contains a shape object ‘‘has a’’ shape object.

To introduce the important concept of *polymorphism*, lets go back to the procedural approach and see how we would write some code involving the drawing of different shapes (in C-like notation):

```
SetUpWindow();
while (...) {
   int shape = chooseShape();
   GetPosition(x,y);
   switch (shape) {
      case 1: DrawRect(x,y);   break;
      case 2: DrawSquare(x,y); break;
      case 3: DrawLine(x,y);   break;
   }
}
```

In the above example we first set up the display window. Next we enter a `while` loop where we get back the shape type selected by the user and the desired drawing position, followed by a `switch` statement for calling the correct draw function. We note that the procedural program’s control structure must be altered to add another type of drawing. Suppose we wanted to draw a triangle. We would have to modify the internal structure of the program in addition to adding a new procedure to draw triangles. This is because of the `case` selection at the core of the program. Modifying the `case` selection is tantamount to redesigning the program.

[1]In this article all class names used in the examples start with a ‘‘T’’, for ‘‘Type’’.

Now lets look at the OO version of the above code fragment (in C++ notation):

```
SetUpWindow();
while (...) {
   TShape *shape = ChooseShape();
   GetPosition(x,y);
   shape->Draw(x,y);
}
```

The control structure looks almost the same – the differences are subtle but important. The most important difference is the elimination of the `case` selection structure. In the `ChooseShape()` function, in response to the user's input, the desired shape is created (e.g. a rectangle, a square, a line) and a pointer to the abstract base class `TShape` is returned[2]. The `Draw()` method is applied to whatever shape is created by `ChooseShape()`. But the shape is not known until the program runs. We call this *dynamic binding*, because the method is bound to the object at the latest possible moment, i.e. when the program is running. The `Draw()` method is executed by the appropriate object so that it can draw itself. We say that the `Draw()` method is *polymorphic* because it takes on different implementations at different points in the program. If the object created by `ChooseShape()` is a rectangle, the `Draw()` method is bound to the `TRect` object's `Draw()` method, if it is a line it is bound to `TLine`'s `Draw()` method. In C++ polymorphism is implemented by declaring a member function to be *virtual*.

Benefits of OOP

In summary, OO programming is a large improvement over procedural programming, because:

- The code is made more general and flexible because `case` selection is replaced by dynamic binding, resulting in smaller, simpler code.
- A greater degree of encapsulation means greater opportunity for reuse.
- Maintenance is easier because all `Draw()` method calling is centralized and rarely needs to be changed. Polymorphism means we can write different `Draw()` methods for different objects, without having to change the calling routine.
- Complexity is reduced because there is little growth of the global state. The state is contained within each object, rather than scattered through the code in the form of global variables.
- Objects may come and go, but the basic structure of the program remains relatively static. This opens up opportunities for the reuse of design as well as code.
- Encapsulation enforces data abstraction: the organization of data into small, independent objects that can communicate with each other. Encapsulation

[2]Objects of derived classes can be referenced via their base class, since they all share the common protocol of the base class.

protects the data in an object from accidental damage, but allows other objects to interact with the data by calling the object's member functions.

- Subclassing and inheritance make it possible to extend and modify objects: that is, to derive new kinds of objects from the standard classes available in the system, thereby creating new capabilities without having to start from scratch.

- Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.

Framework Fundamentals

A *framework* is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behaviour, like ROOT's histograms or Be's menus and windows, and programmers use it by inheriting some of that default behaviour and overriding other behaviour so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

- **Behaviour versus protocol**. Class libraries are essentially collections of behaviours that you can call when you want those individual behaviours in your program. A framework, on the other hand, provides not only behaviour but also the protocol or set of rules that govern the ways in which behaviours can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.

- **Don't call us, we'll call you**. With a class library, the code the programmer writes instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework – that is, to treat the framework as a class library – but to take full advantage of a frameworks reusable design, a programmer writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing up responsibilities among the various pieces of software that get called by the framework rather than specifying how the different pieces should work together. This relationship is expressed by the principle: ''Don't call us, we'll call you''.

- **Implementation versus design**. With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related classes work together. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

A framework often embodies specific domain expertise. Application code customizes the framework in a way that solves a particular application problem. In the case of a framework like ROOT, the expertise is in the domain of large scale scientific data analysis.

Calling API Versus Subclassing API

Frameworks can be thought of as having two Application Programmer Interfaces (APIs): a *calling API*, which resembles a class library, and a *subclassing API*, which is used for overriding framework member functions. The calling/subclassing distinction describes the mechanism by which functions are invoked: call or be called. Figure 2 provides a simplified picture of how programmers use these APIs.

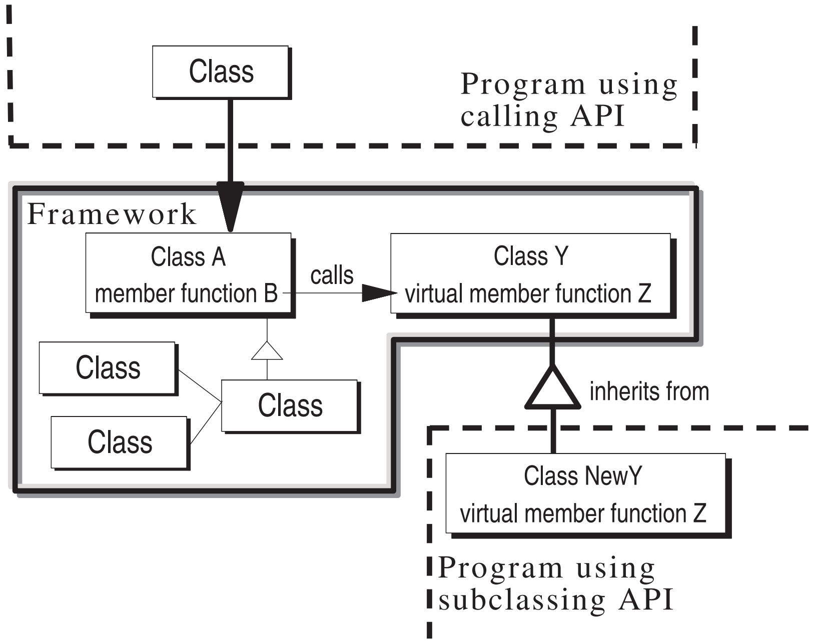

Figure 2 Calling API versus subclassing API

The framework in the center of Figure 2 consists of multiple classes that cooperate intimately to address a particular problem domain. The box above the framework represents a program using the calling API. This program could be another framework or any program that requires the services of the framework in the center of the figure.

For example, the histogramming framework of ROOT allows applications to create, fill and analyze 1D, 2D and 3D histograms by using a single calling API that resembles a class library. Programmers can instantiate histogram objects and use them to histogram any desired quantity.

The box underneath the framework in Figure 2 represents a program using the subclassing API. The subclassing API looks very different from the calling API. A programmer uses the subclassing API by subclassing existing classes and overriding particular member functions. The resulting code is called by the framework when it calls those member functions polymorphically.

In the histogramming example, the subclassing API consists of virtual member functions for a generic implementation of the histogramming interface. The appropriate innards of the generic interface are exposed, via the subclassing API, so that they can be customized for different types of histograms. Because the calling API is the same for different histograms that are subclassed via the histogram subclassing API, programs that use the calling API work the same way no matter which subclassed histogram is actually providing the services. This is the essence of polymorphism.

Benefits of Frameworks

The benefits of frameworks can be summarized as follows:

- Less code to write. Much of the program's design and structure, as well as its code, already exist in the framework.
- More reliable and robust code. Code inherited from a framework has already been tested and integrated with the rest of the framework.
- More consistent and modular code. Code reuse provides consistency and common capabilities between programs, no matter who writes them. Frameworks also make it easier to break programs into smaller pieces.
- More focus on areas of expertise. Users can concentrate on their particular problem domain. They don't have to be experts at writing user interfaces, graphics, or networking to use the frameworks that provide those services.

Conclusions

The ease with which an individual programmer can learn to use a framework depends on that person's previous Object-Oriented design experience, knowledge of framework programming and knowledge of C++ and OOP. After mastering a few frameworks, most programmers find new ones much easier to learn. The long-term benefits of frameworks far exceed the initial difficulties involved in learning this new way of programming.

In the next article I will describe in detail the workings of the Be [1] application building framework and the ROOT [2] scientific data analysis framework. Using a number of concrete examples I will show the power of these frameworks.

Bibliography

[1] Be Inc., *The BeOS*, Be Inc.,1997. See also http://www.be.com/.

[2] Rene Brun, Nenad Buncic, Valery Fine & Fons Rademakers, *The ROOT System*, CERN/HP, 1997. See also http://root.cern.ch/.