OBJECT-ORIENTED DESIGN AND IMPLEMENTATION

Makoto Asai
Hiroshima Institute of Technology, Hiroshima, Japan
(Geant4 Collaboration)

Abstract
This lecture introduces some basic concepts of Object-Orientation and designing and implementing a program based on Object-Orientation. These concepts are more important than the detailed syntaxes of a language and they will guide you to learn C++/Java as a language which stands on Object-Orientation.

1. INTRODUCTION

If you are writing your code which is exclusively used by yourself and it will be used within a temporary short duration, you can ignore the quality of your code. But you are developing your code with your colleagues and/or your code will be used by your collaborators for years, you should be aware of “good software”. Good software is
- Easy to understand the structure
- Easy to find/localize/fix a bug
- Easy to change one part without affecting to other parts
- Well modularized and reusable
- Easy to maintain and upgrade
- etc.
Object-Orientation is a paradigm which helps you to make a good software.

Use of so-called “Object-Oriented language” such as C++ or Java does not guarantee the Object-Oriented Programming. Badly written C++/Java code is worse than badly written Fortran code. Well-designed Object-Oriented good software can be relatively easily implemented by using Object-Oriented language. In this sense language is a tool to realize Object-Orientation. This lecture introduces some basic concepts of Object-Oriented Programming. These concepts are more important than the detailed syntaxes of a language and these concepts will guide you to learn C++/Java as a language which stands on Object-Orientation.

Object-Oriented Programming (OOP) is the programming methodology of choice in the 1990s. OOP is the product of 30 years of programming practice and experience.
- Simula67
- Smalltalk, Lisp, Clu, Actor, Eiffel, Objective C
- and C++, Java

OOP is a programming style that captures the behavior of the real world in a way that hides detailed implementation. When successful, OOP allows the problem solver to think in terms of the problem domain. Three fundamental ideas characterize Object-Oriented Programming.
- Class/Object, Encapsulation
- Class hierarchies, Inheritance
- Abstraction, Polymorphism

In this lecture, it must be noted that all sample codes are written in C++.

2. CLASS/OBJECT AND ENCAPSULATION

2.1 Class and object

Object-Oriented Programming (OOP) is a data-centered view of programming in which data and behavior are strongly linked. Data and behavior are conceived of as classes whose instances are objects. OOP also views computation as simulating behavior. What is simulated are objects represented by a computational abstraction.

The term abstract data type (ADT) means a user-defined extension to the native types available in the language. ADT consists of a set of values and a collection of operators and methods that can act on those values. Class objects are variables of an ADT. OOP allows ADT to be easily created and used. For example, integer objects, floating point number objects, complex number objects, four momentum objects, etc., all understand addition and each type has its own way (implementation) of executing addition. An ADT object can be used in exactly same manner as a variable of native type. This feature increases the readability of the code.

FourMomentum a, b, c;
   c = a + b;

In OOP, classes are responsible for their behavior. For example, the FourMomentum class should have the following definitions to ensure the above equation could be implemented.

```
class FourMomentum
{
    public:
        FourMomentum(double px, double py, double pz, double e);
        ~FourMomentum();
    public:
        FourMomentum& operator = (const FourMomentum & right);
        FourMomentum operator + (const ThreeMomentum & right);
        ....
```

2.2 Encapsulation

Encapsulation consists of the internal implementation details of a specific type and the externally available operators and functions that can act on objects of that type. The implementation details should be inaccessible to client code that uses the type. It is mandatory to make data members private and provide public Set/Get methods accessible to them. Also, make all Get and other methods which do not modify any data member “const”. The “const” methods can be accessed even for constant ADT objects. Strict use of constant ADT objects allows you the safe programming.

Changes of the internal implementation should not affect on how to use that type externally. Following two implementations of FourMomentum should be used exactly same manner from any other code. Figure 1 is an example class diagram taken from Geant4 toolkit[1]. G4Track and G4Step are the classes which hide actual data quantities and provide some public methods to the respective private data members.
3. CLASS HIERARCHIES AND INHERITANCE

Inheritance is a mean of deriving a new class from existing classes, called base classes. The newly derived class uses existing codes of its base classes. Through inheritance, a hierarchy of related types can be created that share codes and interfaces. A derived class inherits the description of its base class. Inheritance is a method for copying with complexity. Figure 1 is an example of class hierarchy.

It is better to avoid protected data members. Data members in a base class should be private and protected non-virtual access methods to them should be provided. Also, it is advised to avoid
unnecessary deep hierarchies. For example, should a trajectory class and a detector volume class be derived from a single base class, even though both of them have a “Draw()” method? Following the naïve concepts that everyone can easily understand leads the well-designed and thus easy-to-maintain code. Also, it is advised to avoid unnecessary multiple inheritance. In many cases, delegation can solve the problem. Figure 3 is an example of delegation.

Type-unsafe collection is quite dangerous.
- In C++ case, pointer collection of void or very bogus base class.
- In Java case, default vector collection of “Object” base class.

Type-unsafe collection easily reproduces the terrible difficulties we experienced with the Fortran common block.

Figure 2. A class diagram taken from Geant4 toolkit which demonstrates class hierarchy

Figure 3. A class diagram taken from Geant4 toolkit which demonstrates delegation
4. ABSTRACTION AND POLYMORPHISM

Abstraction and Polymorphism enable “Rapid Prototyping”. High level class diagrams and scenario diagrams should be made first before going to the detailed design/implementation of actual concrete classes. Proof of concepts demonstration must be done with just a couple of concrete classes (or just one dummy concrete class) for each abstract base class.

Abstraction and polymorphism localizes responsibility for an abstracted behavior. They also help the modularity and portability of the code. For example, Geant4 is free from the choice of histogramming or persistency techniques. Also, GUI and visualization in Geant4 are completely isolated from Geant4 kernel via the abstract interfaces.

Polymorphism has lots of forms.
- Function and operator overloading
- Function overriding
- Parametric polymorphism

In C++, an operator is overloadable. A function or an operator is called according to its signature, which is the list of argument types. If the arguments to the addition operator are integral, then integer addition is used. However, if one or both arguments are floating point, then floating point addition is used. Operator overloading helps the readability.

```cpp
double p, q, r;
r = p + q;
FourMomentum a, b, c;
c = a + b;
```

Using virtual member functions in an inheritance hierarchy allows run-time selection of the appropriate member function. Such functions can have different implementations that are invoked by a run-time determination of the subtype, which is called virtual method invocation or dynamic binding.

```cpp
G4VHit* aHit;
for(int i = 0; i < hitCol->entries(); i++)
{
    aHit = (*hitCol)[i];
aHit->Draw();
}
```

In this code sample, “Draw()” method is virtual. Thus the way how actual hits will be drawn is not (or should not) known at this code. The way should differ for each type of hit (e.g. calorimeter or drift chamber) and it should be implemented in the individual hit concrete class.

Functions of same name are distinguished by their signatures. For the case of function overloading of “non-pure virtual” virtual functions, all (or none) of them should be overridden. It must be noted that overriding “overrides” overloading. This is an intrinsic source of a bug even though compiler warns. Following sample gives an output of “Int” instead of “Double”.

```cpp
double p, q, r;
r = p + q;
FourMomentum a, b, c;
c = a + b;
```

class Base {
    public:
    Base() {}
    virtual void Show(int i) { cout << “Int” << endl; }
    virtual void Show(double x) { cout << “Double” << endl; }
}

Class Derived : public Base {
    public:
    Derived() {}
    virtual void Show(int i) { cout << “Int” << endl; }
}

main() {
    Base* a = new Derived();
    a->Show(1.0);
}

C++ also has parametric polymorphism, where type is left unspecified and is later instantiated. STL (Standard Template Library) helps a lot for easy code development.

5. UNIFIED SOFTWARE DEVELOPMENT PROCESS

A software development process is the set of activities needed to transform a user’s requirements to a software system (see figure 4). The Unified Software Development Process (USDP)[2] is a software development process which is characterized by Use-case driven, Architecture centered and Iterative and incremental.

There are many different types of requirements.

- Functional requirements
- Data requirements
- Performance requirements
- Capacity requirements
- Accuracy requirements
- Test/Robustness requirements
- Maintainability, extensibility, portability, etc., “ability” requirements

All of these requirements drive use-cases and architectures.

![Figure 4. Software process](image-url)
A software system should be used by the user. Thus the developer of the system must know the users’ needs. The term user refers not only to human users but also to other system which interacts with the system being developed. An interaction from/to the user is a use-case. A use-case is a piece of functionality in the system which captures a requirement.

The role of software architecture is similar in nature to the role of architecture plays in building construction. A plan of building is looked at from various viewpoints, such as structure, services, heat conduction, electricity. This allows the builder to see a complete picture before actual construction. The software architecture must be influenced by the requirements of both use-case dependent and use-case independent. Architecture is not a framework.

The Unified Modeling Language (UML)[3] has various diagrams. Not the all diagrams must be used for the software development. But it must be stressed that many of them are quite useful for the Object-Oriented software design and implementation. In this lecture, details of UML diagrams are not covered but the lecture in this school given by R. Jones[4] gives the detail.

A software process has four steps, Object-Oriented Analysis, Object-Oriented Design, Implementation and Test. These steps must be taken many turns (so-called spiral approach) instead of once (so-called water fall approach). The first turn should be the “proof of concepts” demonstration. User’s requirements document and diagrams should be regularly updated during the turns.

6. SUMMARY
Basic concepts of Object-Oriented Programming are more important than the detailed syntaxes of a language and these concepts will guide you to learn C++/Java as a language which stands on Object-Orientation. Well-designed, Object-Oriented good software can be relatively easily implemented by using Object-Oriented language.

ACKNOWLEDGEMENTS
The author wishes to thank the organizers of the school for their supports on this lecture.

REFERENCES