



Is Schrödinger's Cat Object-Oriented?

By Adolfo M. Nemirovsky

Why Natural Scientists Should Care About Object-Oriented Technology

Object-Oriented technology is gaining rapid acceptance among software developers, and is becoming the preferred choice for modern computer programming projects. Should a natural scientist care? We discuss some of the main concepts in object-oriented programming and the potential of this interesting technology. The object model views the world as made of many objects interacting (exchanging messages) with each other to produce a collective behavior. This picture resembles a quantum system of interacting particles. Suggestive analogies between the object model and quantum physics are identified and exploited in this work to provide an introduction to object-oriented programming

This paper is directed to natural scientists and graduate/advanced undergraduate students in physical sciences. Some basic concepts of quantum physics at the level of introductory courses in modern physics or physical chemistry is assumed. Background in quantum mechanics is helpful but not required to understand most of this work. No prior knowledge of C/C++ and object oriented programming is assumed. Most of the C++ code illustrating some of the physics examples presented in the paper is given in the Appendices.

Table of Contents

- I. Why Object-Oriented?**
- II. Basic Concepts in the Object Model**
- III. The Object Model and Quantum Physics**
- IV. Classes, Hierarchies and Quantum Field Theories**
- V. Dynamics: Objects, Messages and Feynman Diagrams**
- VI. Object-Oriented Design and Effective Hamiltonians. Frameworks**
- VII. Multimedia, Parallel Universes and Path Integrals.**

VIII. Present and Future

Acknowledgments

APPENDIX A: Writing C++ Code I: The Class TVector3D

APPENDIX B: Writing C++ Code II: TParticle, TNucleon and TProton

Classes

References

Copyright

I. Why Object-Oriented?

Object-Oriented (OO) technology is becoming one of the most important technologies in software development. It has enormous potential for significantly increasing programmer productivity and code maintainability.[1-3] Why is OO gaining such a widespread acceptance among commercial software developers and large corporations? Procedural code of industrial strength (thousands of lines of code) is often unstable under small perturbations, e.g., trying to fix a single problem might create five new ones[2,3] When the code size is larger than some threshold (about 10^5 lines of code), procedural techniques appear to break down: design and implementation complexity makes projects quite difficult and expensive, and eventually maintenance cost dominates development cost[2,3] Object-Oriented techniques promise to allow more stable and larger applications (several millions and tens of millions of lines of code) to be built[1-3] How can it be done? Why should a natural scientist care?

Natural scientists model and simulate natural phenomena. Computers simulate "real" (e.g., balancing a checkbook, mimicking the behavior of a physical system,...) and "imaginary" (e.g., games) worlds. Rapid advances in hardware and software are making possible the treatment of interesting and complex systems (e.g., turbulent flows, critical dynamics, lattice quantum chromodynamics, weather prediction, cellular metabolism, social phenomena,...). Simula, a language that pioneered object-oriented technology was specially conceived for simulations. Hence, it is not surprising that object-oriented programming may make simulation and visualization of complex phenomena easier than procedural techniques (e.g., Fortran). For fully exploring models of complex systems, displaying results and data in a powerful manner (e.g., scientific visualization, multimedia, virtual reality,...), for enhancing collaboration fully exploiting networking facilities, OO technology appears better suited than procedural languages.

The principle "divide and conquer" has been known for a long time as an efficient technique to conquer large empires (e.g., large codes). But, how to divide? Both procedural and OO programming employ the principle. The fundamental difference between these two models is the choice of building blocks. In procedural programming, procedures are the fundamental units (usually called subroutines or subprograms). In the object model, basic units are "cells" or "atoms" called objects which contain both data (i.e., state variables associated with the "atom" state at a given time) and methods (i.e., dynamical rules, rules that explain how "atoms" interact in the outside world). In OO terminology, objects encapsulate data and behavior. These objects are individual weakly interacting blocks. Objects interact (exchange messages) to produce collective behavior.

OO views the word as composed of objects with well-defined properties. Object

dynamics is pictured as interaction among objects. Interactions are mediated by messages that objects exchange with each other. In fact, for a physicist this description may sound familiar as it resembles our modern view of the atomic and subatomic world. This suggests that we could expect interesting analogies between Quantum Physics (QP) and the object model (OM). These analogies are identified and exploited in this work to present an introduction to the OO philosophy. Our examples use C++ as it currently is the most widely used OO language.[4].

The main goal of this work is to provide an introduction to OO concepts for an audience of natural scientists and graduate/advanced undergraduate students in the Physical Sciences. We introduce some of the main OO ideas in this work at three different levels. At the highest level, we discuss general OO concepts, independent of the language. At an intermediate level, and to produce more concrete examples, we express some of the QP concepts[5] in the C++ language. [C++ code is written in this style, and adopts the Taligent notational style[6]]. Usually, in this process we select a particular C++ design, the simplest to illustrate some OO features. Selected designs rarely are the "best" and usually oversimplify the physics. Thus, the code provided in this work is mainly for didactic purposes and not intended for a real project.

Section II presents some basic concepts in the object model (OM): abstraction, encapsulation, modularity and hierarchy. They are illustrated with examples from the atomic/subatomic world. Connections between the OM and some concepts in Quantum Physics are presented in **Section III**. **Table 1**, at the end of that section, summarizes some of these analogies. **Section IV** discusses additional analogies between structures in quantum physics and constructs in the object model, and then summarizes them in **Table 2**. Dynamics in the OM and in QP are discussed and contrasted in **Section V**. **Section VI** introduces the problem of object-oriented design of a complex system. Analogies between hypermedia and the path integral formulation of QP are presented in **Section VII**. Finally, the **last section** collects some final thoughts and summarizes some of the experiences of earlier scientific users of the object-oriented technology.

II. Basic Concepts in the Object Model

The *object model* provides theoretical foundations upon which object-oriented design is built. The OM is based[1] on the principles of abstraction, encapsulation, modularity and hierarchy, introduced in this section and expanded in the rest of the paper.

Abstraction is well known to any natural scientist or anybody interested in modeling. It consists of extracting the relevant common features of the system to be modeled, providing adequate generalization, and removing irrelevant details. For example, all protons in the universe share many properties: value of their masses, electric charges, spins,... (data) and the way they interact with themselves and other particles (methods). Protons are still protons, regardless of their location (inside our mouth or in a distant galaxy) or their internal state (i.e., value of their momentum, orientation of their spin,...).

Abstraction determines the characteristics of an object that distinguish it from other kinds of objects (e.g. protons are different from neutrons or electrons, but two protons in different atoms are still protons). This is the "outside view" of the object. That is how clients, particles that interact with the proton, see it. It is important to have the right set of abstractions for a given problem domain. For example, to different levels of abstraction, protons and neutrons can be either viewed as different objects, or the

same object (nucleon) in a different state. We will return to this later. There is a wide spectrum of abstraction possibilities: objects could closely resemble "real" objects (car object, pendulum object,...) but there could be quite "abstract" objects with apparently "no reason to exist" (shape object, transaction object, photon, Cooper pair,...).

Encapsulation is also called *information hiding*. Clients (particles that interact with protons) do not see a proton's internal structure ("inside view" of the object). That is, protons hide information about their internal structure. In general, the interface of a class captures only the outside view. For example, electrons of hydrogen atoms do not know the proton internal structure. They only see a proton's interface: its mass, electric charge, linear momentum and spin. Encapsulation, then, is the process of hiding all the internal details of an object.

Modularity means that a system can be decomposed into a set of weakly coupled modules. Hydrogen gas is composed of many loosely interacting objects (hydrogen molecules). Each molecule is made of two weakly interacting objects (hydrogen atoms). In turn, each hydrogen atom consists of two objects (proton and electron) that collaborate (are bound together) exchanging messages (photons, the messengers, are the carriers of the electromagnetic interaction). In general, one of the most difficult problems in the OO design of complex systems is to "discover" the right modules to mimic the system. This difficulty is analogous to finding the "right" ground state in a many-body problem (e.g., superconductivity) as discussed later.

Hierarchy is the ranking or ordering of abstractions. Consider a deuterium atom. It has just a single electron, but the nucleus consists of a neutron and a proton. Electrons in the deuterium atom interact (exchange messages = photons) with charged protons through the electromagnetic interaction. Neutrons are uncharged particles so, at this level of abstraction, electrons see neutrons and protons as different particles (objects). At a higher level of abstraction, now inside the deuterium nucleus, protons and neutrons "collaborate" by sending "attractive" messages to each other (exchanging pions). They view each other as different states of the same particle (in physicists language, they are members of a SU(2) doublet). At still higher levels of abstraction, nucleons (protons or neutrons) are just possible states of more abstract entities (SU(N) multiplets, with $N = 3, 4, \dots$) that encompass a large number of distinct particles at lower levels of abstractions. Fig. 1 displays this example of subatomic hierarchical layering.

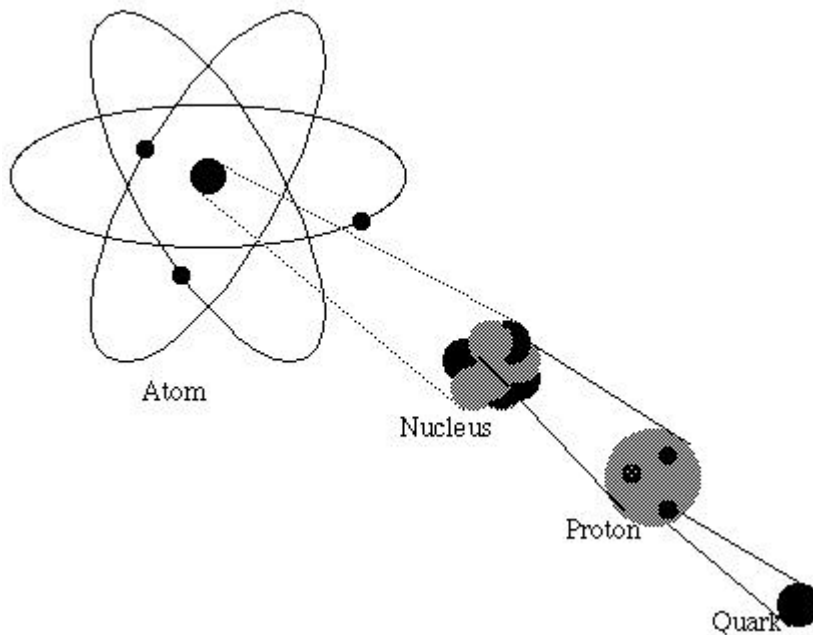


Figure 1. Subatomic hierarchical layering: in this picture we distinguish three levels of abstraction. In atomic physics, layer 1, we describe the physics in terms of interacting electron and nucleus objects. Inside the nucleus, layer 2, the appropriate objects are protons and neutrons. Finally, at higher collision energy, layer 3, the proton is viewed as made of interacting quark objects.

III. The Object Model and Quantum Physics

Objects, the basic units in OO design, have *state, behavior and identity*. *Methods* or *member functions* are operations that act upon objects and may modify their states. These concepts are introduced in this section using analogies with QP. Since linear algebra is the mathematical language of quantum mechanics, these analogies between OM and QP are also expressible in the linear algebra language.

In QP[5], the description of the state of a system (say, a particle) at a given time is defined by specifying its state vector. This vector belongs to the state space \mathbf{E} (which comprises all possible states of the system). The set of distinct states of a system can either be finite or infinite (countable or continuous). [Of course, the computer representation of the state space is always finite]. The description of the state of an object in the OM follows from the above by just replacing "system" (or "particle") by "object". Examples:

- 1) Consider an Ising spin or Boolean variable. This object can only take two values: up (on or 1) and down (off or -1). These two distinct states can be represented by the vectors

$$|up\rangle \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |down\rangle \equiv \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

We have used the so-called Dirac notation $| \rangle$ to denote a state vector. This example is easily generalizable for objects with 3, 4,... states.

Given the two states up and down, in QP a state vector could be any linear combination of these two basis vector. The only constraint is that vectors are normalized (i.e., the norm of the vector should be unity). Hence, a more general state vector $|s\rangle$ for the two-state system has the form

$$|s\rangle \equiv \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}, \text{ with } 0 \leq \theta < 2\pi.$$

The interpretation is the following: the spin is in a state with probability $\cos^2(\theta)$ to be $|up\rangle$, and $\sin^2(\theta)$ to be $|down\rangle$. Conservation of probability (i.e., $\cos^2(\theta) + \sin^2(\theta) = 1$) constrains the state vectors to be normalized.

In our C++ design, an object of the type `IsingSpin` is denoted `anIsingSpin`. This object encapsulates data, i.e., the value of the spin +1 or -1, and methods such as `Flip()`, discussed later. In the OO literature usually one assigns either state $|up\rangle$ or $|down\rangle$ to the object, but *not* some linear combination of these two vectors. An object, in the common OM usage, resembles more a "classical" entity than "quantum" one. This is not a limitation of the OM. In fact, the OM is equally suited to model quantum mechanics by assigning a "probabilistic" interpretation to the states of an object.

2) Consider a "classical" three dimensional real vector object. In this case there is a continuum of states. Once a coordinate system is chosen, say, Cartesian, states are labeled by the x, y and z coordinates of the vector. Our vector object can be represented in the Dirac notation as $|x, y, z\rangle$. In our C++ model, we just call it `aVector3D` and model it as a 3-dimensional array. The coordinates x, y and z are the data that the object encapsulates. There are methods such as `RotateZ(theta)` discussed later which act on the object's data and produce a rotated vector.

3) Consider a spinless particle. It is described by its state vector $|k\rangle$ with $\mathbf{k} = (k_x, k_y, k_z)$ the linear momentum coordinates. The quantities k_i , $i=x, y$ and z are real numbers. An alternative description of the same particle might be in terms of the position state vector $|x\rangle$ with \mathbf{x} a three dimensional vector, i. e., $\mathbf{x} = (x, y, z)$. The particle state vector is related to the probability of finding the particle in a particular spacial region (or with a particular value of linear momentum). In our design, we denote the particle `aParticle`. It encapsulates data (say the value \mathbf{k} of the momentum) and methods such as `MomentumTransfer()` given below.

Identity [1,2] is the property that distinguishes an object from all other objects. For example, this proton, now in my mouth, is not the same as (but it is identical to) that proton now at a distant galaxy. In QP each object defines its own vector space spanned by a complete set of linearly independent vectors associated with all its possible states. In the OM common usage, identity of objects is always preserved even when states are completely changed. A proton object is still the same object even if its momentum and spin changes because of interaction with other objects. In other words, identity means that there is a unique state vector space associated with each object. Although in classical physics identity is always preserved, in quantum physics and Nature, sometimes an object could either have a "non-sharp" identity, or even completely lose its identity. This could, but usually is not, be modelled in the OM,[3] and it is further discussed in the following sections.

Behavior^[1,2] is how an object acts or reacts in the presence of other objects, i.e., how it interacts with the external world. Behavior could also be defined as response to actions (i.e., response function). Operations that affect the object and possibly modify its state are called *methods* or, in the C++ terminology, *member functions*. In QP, methods correspond to operators, that is, the state of the system is modified by applying an operator (i.e., a matrix) O on the state vector $|s_1\rangle$, producing a new vector $|s_2\rangle$ as a result, i.e.,

$$|s_2\rangle = O |s_1\rangle ;$$

The above operation might be written in the OM language (employing C++ notation) as

```
myObject.SetState(s2);
```

where `myObject` is an object whose initial state is `s1`, and that makes a transition to `s2` after applying the method `SetState()`. This is an oversimplified representation as the `SetState()` method is independent of the initial state `s1`. A better design might be

```
myObject.ChangeState(r);
```

where $r = s_2 - s_1$ is the "distance" between the states `s1` and `s2`.

In QP, operators are linear (i.e., represented by matrices) acting on linear vector spaces. On the other hand, in real life OO design, operators often are not linear. OO designs commonly employ operations that either change the state of objects (for example, *set-methods*) or access their state (*get-methods*). Get-methods measure some object's observables (or in the OO language, read the data encapsulated by the object). Besides these, there are two very important methods called *constructor* and *destructor*, which create and annihilate objects. As it could be guessed, they are closely related to the creation and annihilation operators of second quantized quantum mechanics, and they are discussed in the following section. Now we provide some examples,

1) For `anIsingSpin` object, we might have the following methods (operators)

$$\text{Flip}() \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \text{SetUp}() \equiv \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \text{SetDown}() \equiv \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}, \text{GetSpin}() \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The method `Flip()` changes the state of the system, i.e., switches the state of the spin object (i.e., up <-> down). In our C++ design we write

```
anIsingSpin.Flip();
```

where `anIsingSpin` is an object of the type (class) `TIsingSpin`, initially in some state, say $|up\rangle$. Application of the method `Flip()`, flips the spin so now `anIsingSpin` is in the state $|down\rangle$. An alternative, matrix representation of this operation (assuming the initial state of `anIsingSpin` was up) is

$$\text{anIsingSpin.Flip}() \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} .$$

The methods `SetUp()` and `SetDown()` (set-methods) set the state of the spin to up and down, irrespective of the previous state. Finally, the eigenvalues of `GetSpin()` provide the actual values of the spin (either up or down). In other words, $|up\rangle$ and $|down\rangle$ are eigenvectors of the `GetSpin()` operator with eigenvalues 1 and -1, respectively. In general, in the object model (and more general modeling) it might not be possible to define Get operators (as in QP) such that their eigenvalues provide the only possible results of measurements.

2) For the `TVector3D` object one can define, among others, the following method

$$\text{RotateZ}(\theta) \equiv \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which rotates any `TVector3D` object, given in Cartesian coordinates, counterclockwise an angle θ around the z axis. In C++ notation, this operation can be written as

```
aVector3D.RotateZ(theta);
```

where the object `aVector3D` is of type `TVector3D` (that is, is a real three dimensional vector). After applying the method `RotateZ(theta)` to `aVector3D`, it changes state, rotating an amount (θ) around the z axis.

3) The linear momentum operator P [denoted `GetMomentum()` in our design] accesses the state of the proton (say, `aProton`) and reads its data,

$$P | \mathbf{p} \rangle = p | \mathbf{p} \rangle ,$$

in C++ this reads,

```
aProton.GetMomentum();
```

Usually one also defines a momentum transfer operator, `MomentumTransfer()`, which when acting on `aProton`, changes the value of its momentum from $\mathbf{p1}$ to $\mathbf{p2}$ (alternatively, in a simpler model we might have defined a `SetMomentum()` method). In this note, and for simplicity we do not provide any mathematical representation of the momentum state vectors and operators, and just use the more general (basis independent) Dirac notation [6].

Here we summarize the correspondence between OM, QP and linear algebra :

TABLE 1. The Object Model, Quantum Mechanics and Linear Algebra

Object Model	Quantum Mechanics	Linear Algebra
an object	a quantum particle	
state	state vector	vector
method/member function	observable/operator	matrix
behavior	response function	
encapsulation	state vector space	vector space

IV. Classes, Hierarchies And

Quantum Field Theories

In the OM we think of objects as concrete entities that exist in space/time. In contrast, a *class* is an abstraction which summarizes objects' common properties (structure and behavior). Each object is always some instance of a class. For example, the class `TProton` describes all abstract properties shared by proton objects. The concrete object `aProton` is an instance of the class `TProton`. Both, the OM and QP are not only interested in describing a single object or particle but also many mutually interacting ones. In this section we compare the OM and the QP descriptions of systems of many identical objects. We also present a design of the class `TProton`. In addition, using C++ notation, we introduce several important OM concepts such as base and derived classes, simple and multiple inheritance, and polymorphism.

Let us first discuss the QP description of a system of many identical particles. Consider a system of two `TSingSpin` objects labeled `anSingSpinA` and `anSingSpinB`. Each one has, associated with it, a 2-dimensional vector space $E(a)$ and $E(b)$, spanned by the vectors up and down for both objects, `anSingSpinA` and `anSingSpinB`. To the composite system corresponds a vector space $E = E(a) \times E(b)$ (\times indicates the direct product of the two vector spaces $E(a)$ and $E(b)$) spanned by four vectors $|A \text{ up} \rangle \times |B \text{ up} \rangle$, $|A \text{ up} \rangle \times |B \text{ down} \rangle$, $|A \text{ down} \rangle \times |B \text{ up} \rangle$, $|A \text{ down} \rangle \times |B \text{ down} \rangle$, corresponding to all possible combinations of states of the spins A and B. Obviously, this discussion is easily generalizable to many objects, say `anSingSpin(i)` with $i=1,2,\dots, N$. The composite system made of N objects of type (class) `TSingSpin` has, associated with it, a state vector space E , the direct product of the individual state vector spaces $E(i)$ for all i .

One-body operators, such as `SetUpA()`, only operate on `anSingSpinA` setting its state to up, but do not affect other objects. Mathematical representations of a given vector depend on the choice of coordinates, i.e., a vector has many possible representations. For a system of many particles there is an alternative (equivalent) representation of the system's state vector which does not hinge on the individual state vectors but focuses on the system's properties such as total spin (sum of the spin of individual particles), total number of particles, total energy/momentum,.... This alternative representation of the same physics known as second quantization [5] is closely related to a "class view" in the object model. Although second quantization is the preferred formulation of the non-relativist quantum mechanical many-body problem, it is the only choice when the fully relativistic extension is considered. This is because in the relativistic view there cannot be a single (or a few) object simplification of reality but there must always be a multitude of coexisting objects.

An important feature of the second quantization formalism is the existence of the creation and annihilation operators. These operators, are closely associated with the constructor and destructor operators common in most OO languages (such as C++). In the second quantization formalism, *Proton becomes a quantum field*. Associated to it, there is a creation and an annihilation operator. Similarly, in the OM *TProton becomes a class*. Associated to it, there are constructors and destructors. Using C++ notation, the creation (constructor) and annihilation (destructor) operators are denoted `TProton()` and `~TProton()`, respectively. [In C++ there can be several constructors.] State vectors are now associated to systems with zero, one, two,...., particles (quanta) each in one of its possible allowed states. The null particle system is known as the vacuum (i.e., a class that has not been instantiated). Contrary to one's naive expectation, the null particle state could be quite complex and its effects may be observable.

Creation operators and constructors serve very similar purposes. For example, in C++ one instantiates (creates and initializes) the class `TProton`, creating an object (particle) of the class, which we call `aProton`

```
TProton aProton;
```

In the QP language `aProton` is a one-particle state (a single quantum of the Proton quantum field). The constructor above is the so-called default constructor as it does not take any argument (it might use a random number generator to select a value for the proton's momentum). Now suppose that we would like to create another proton with a given momentum, `aMomentum = (aMomentumX, aMomentumY, aMomentumZ)`. We would then design a constructor that takes an argument, i.e., the value of the momentum: `aMomentum`,

```
TProton bProton(aMomentum);
```

`aMomentum` is an object (real three dimensional vector) of the class `TVector3D`. Before using it above, it might be instantiated as follows

```
TVector3D aMomentum(aMomentumX,aMomentumY,aMomentumZ);
```

where `TVector3D(x,y,z)` is one of the constructors of the class `TVector3D` and takes the three arguments in parenthesis to be the values of the Cartesian components of the vector object to be created (see [Appendix A](#)).

Constructors instantiate objects. Using the constructor we can instantiate as many proton objects as desired. Suppose we would like to create an oxygen nucleus system that contains eight protons and eight neutrons with arbitrary (randomly chosen) momenta, we could proceed as follows

```
TProton aProton[8];  
TNeutron aNeutron[8];
```

It is usual practice in C++ to declare a class, say `TProton`, in a file called a header file while implementation of functionalities is placed in one (or several) source files. Clients of `TProton` usually will need the header file, `Proton.h`, (interface of `TProton` with the exterior world), and seldom, if the design is correct, would information about actual implementation be required. The declaration of the class `TProton` follows

```
#include <Vector3D.h>  
  
class TProton {  
  
public:  
  
        TProton          ();  
        TProton          (TVector3D aMomentum);  
  
        virtual          ~TProton          ();  
  
        virtual void      SetMomentum      (TVector3D aMomentum);  
  
        virtual TVector3D  GetMomentum      ();  
  
        virtual long       GetNumProtons    ();  
  
        virtual double     GetMass          ();  
  
        virtual long       GetElectricCharge ();  
  
private:  
  
        static long       fgNumber;  
  
        static const double kMass;  
  
        static const long  kElectricCharge;
```

```

    TVector3D fMomentum;

};

```

To accomplish *information hiding*, classes have public and private *members*. (In fact, there are also protected members, as discussed in the [Appendix B](#)). *Members* include both data members (fgNumber, kMass, ...) and member functions or methods (constructors/destructor, get/set methods,...). Public members are accessible to any client that uses the class definition (i.e., includes the header Proton.h describing TProton's interface). [In turn, the class TProton is a client of TVector3D since it uses objects of this type. Then, the header Vector3D.h (which is given in the [Appendix A](#)) must be included (first line of the above code) allowing TProton access to public members of TVector3D.] The public members of the class TProton are the constructors, the destructor and the set/get methods. Private members can be accessed only by objects of the class, i.e., by protons. They are hidden from other types of objects, such as electrons. Private data members in the class TProton include the value of its mass, electric charge, momentum, and the total number of protons. In the design above, we have chosen to provide public access to the proton's private data through get methods.

Static members act as a global class variables, i.e., all objects of a particular class have access to the same variable. In the class TProton, fgNumber, kMass and kElectricCharge are all static. Besides this, the last two data members are also constant (const) as these values are fixed for all protons. In contrast, fgNumber changes as protons are created or destroyed. One of the weakness of this design is that any proton object "knows" (i.e., can access) the total proton number, fgNumber. Alternative TProton designs are discussed in [Appendix B](#). The private data member fMomentum is an object of the class TVector3D. It is not static as every object owns its own value for fMomentum. Neither is it constant since this value changes as aProton object interacts with other objects.

In [Appendix B](#), we start at a higher level of abstraction, defining the classes TParticle and TNucleon. In that design (see Fig. 2) TNucleon directly derives from TParticle, and in turn, TProton and TNeutron derive from TNucleon. *Inheritance is one of the fundamental characteristics of OO programming*. In C++, it is supported by the mechanism of class derivation. *Derived classes* inherit properties (i.e., data members and member functions) from the parent class (known as the *base class*). Derived classes usually add new members and/or override methods (i.e., provide different implementation). For example, TNucleon is a *base class* relative to TProton and TNeutron. Then, implementation of most functionalities required by derived classes (such as TProton and TNeutron) is already provided by (inherited from) their base class (TNucleon). In [Appendix B](#) design, the class TProton only needs to override the constructors and destructor, and to implement just an additional method: GetElectricCharge(). Notice that the declaration of TProton in [Appendix B](#) is much simpler than that presented above, as there we have taken advantage of inheritance.

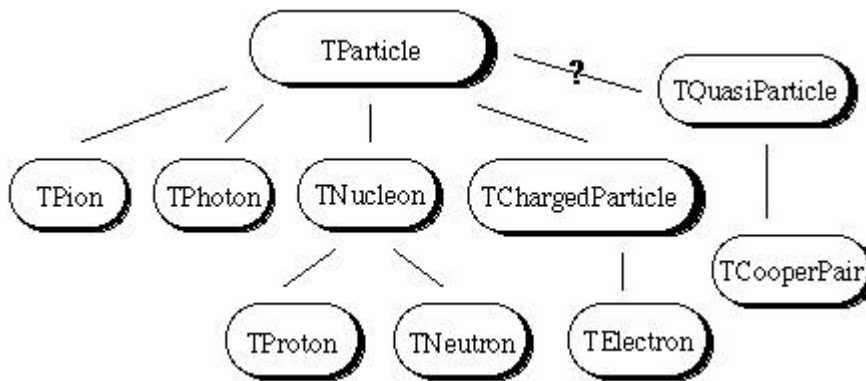


Figure 2. A possible class diagram of the particle "zoo" displaying single inheritance relationship among classes. In this design, TParticle and TChargedParticle are abstract base classes (they cannot be instantiated), all other classes are concrete. Should TPion and TNucleon derive directly from TParticle, or via TChargedParticle? Should TQuasiParticle derive from TParticle, or be at the top of the hierarchy in parallel to TParticle? In OO design of complex systems, the class taxonomy is usually a highly non-trivial problem.

At the highest level of the hierarchical structure (see, Figure 2) we define the class TParticle that encompasses properties shared by all particles (e.g., fMomentum, SetMomentum(), GetMomentum() ...). This class is called *abstract base class* and is never instantiated. That is, we never create quanta of TParticle, but only create objects of its derived classes such as TProton, TElectron, TPion,.... Nevertheless, abstract base classes are useful as they serve to set a common protocol (common notation for all derived classes). Figure 2 presents a possible *single inheritance* class diagram of the particle "zoo". Protons are fermions (the value of the proton's total spin is half-integer, i.e., 1/2); and they are also charged particles. At higher levels of abstraction we might define the abstract base classes TFermion and TChargedParticle. In this higher abstraction, TProton would inherit from both base classes. This is an example of *multiple inheritance*.

Inheritance is just one (although a very important one) of the possible relationships among classes. It is also known as "kind of" relationship since, say, TProton is a "kind of" TNucleon . Another important class relationship in OO design is "part of". For example, protons are made of three quarks. Then, aQuark[3], an array of three quark objects, is "part of" a proton. For example, a possible TProton class design might include

```
TQuark aQuark[3];
```

say, as a private data member. This design makes more explicit some aspects of the proton's internal structure which may be required in some problem domain. Which is the "best" class design and the "best" class taxonomy? The answer strongly depends on the piece of reality we try to capture and the questions we intent to address. These are some interesting issues in OO design further discussed in [Section VI](#).

We now introduce polymorphism which we exploit in the coming section. *Polymorphism* means that a single name may denote methods of many different classes related by some common base class (these methods might act differently on objects belonging to different classes). For example, we might provide the method GetElectricCharge() in the class TChargedParticle . A client's code then works correctly for, say, objects of the classes TProton, TElectron, ..., (returning +1, -1, ...) which derive from TChargedParticle even if the compiler only knows these objects are of type TChargedParticle. That is, code works correctly even if actual values of particles' electric charges are only known at runtime. Polymorphism combines the features of inheritance with dynamic binding (that is, the types of variables and expressions is not known until runtime) and is one of the most important concepts in OO.

Finally, to conclude this section we summarize some of the analogies between the OM ("class view") and quantum field theory.

TABLE 2. Classes and Quantum Fields

Class View	Quantum Physics
class	quantum field
object (class instantiation)	quanta
constructor	creation operator
destructor	annihilation operator

V. Dynamics: Objects, Messages And Feynman Diagrams

Dynamics of a many-object system is the result of object interactions. Objects interact with each other exchanging messages thus changing their states as time evolves. What is the OM formalism used to describe this picture and how does it relate to the QP description? This section discusses and compares the OM and QP descriptions of dynamics. Also, the useful property of operator overloading is introduced.

Suppose we would like to describe an electron-proton scattering process. Initially there is an electron object in the state `aMomentum_K1` and a proton object in the state `aMomentum_P1`. After an interaction, which is mediated by a messenger object called `aPhoton`, both objects (electron and proton) transition to different states, say `aMomentum_K2` and `aMomentum_P2`, respectively. In QP the interaction between quantum objects is often displayed in Feynman diagrams

[5,8] For example, Fig. 3 shows a Feynman diagram for the electron-proton scattering process. [In fact, the interaction of Fig. 3 is just one (although the most relevant) of the very many possible ways proton and electron objects can interact electromagnetically exchanging momentum]. How might this be described in the object model?

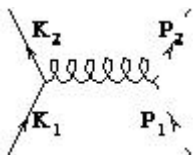


Figure 3. Feynman diagram describing electron-proton electromagnetic scattering. Two objects (proton and electron) in some initial states ($|P1\rangle$ and $|K1\rangle$) interact exchanging a message (a photon). After the interaction, the objects are in different states ($|P2\rangle$ and $|K2\rangle$).

1) Using the appropriate constructors we instantiate an electron, `myElectron`, and a proton,

myProton, with momenta aMomentum_K1 and aMomentum_P1

```
TElectron myElectron (aMomentum_K1);
TProton myProton (aMomentum_P1);
```

2) Using the method `SetMomentum()`, we change the values of the momenta to their final values: aMomentum_K2 and aMomentum_P2. In [Appendix B](#) design, the method `SetMomentum()` is provided in the abstract base class `TParticle`. The classes `TElectron` and `TProton` (which derive from `TParticle`) inherit this method.

```
myElectron.SetMomentum (aMomentum_K2);
myProton.SetMomentum (aMomentum_P2);
```

The values of, say, aMomentum_K2 is constrained due to the momentum conservation law, i.e.,

$$aMomentum_K2 = aMomentum_P1 + aMomentum_K1 - aMomentum_P2;$$

[Note that in the above equation, and in some of the forthcoming equations, we assign, add and subtract `TVector3D` objects as if they were built in types].

Alternatively we might have used the method `MomentumTransfer()` instead of `SetMomentum()` as follows

```
myElectron.MomentumTransfer ((-1)*aMomentum_Q);
myProton.MomentumTransfer (aMomentum_Q);
```

where

$$aMomentum_Q = aMomentum_P2 - aMomentum_P1;$$

Momentum conservation in these simple models must be implemented by hand.

We now present another model that builds in momentum conservation. The above representations only use one-body operators (i.e., `SetMomentum()` or `MomentumTransfer()`, in the class `TProton`, only acts on objects of the type `TProton`). What if we would like to represent a `Photon` object that mediates the electron-proton interaction? A possible way to accomplish this is by using the methods `EmitPhoton()` and `AbsorbPhoton()`, which take as argument an object of type `TPhoton`. The implementation of these methods could easily build in momentum conservation in a natural way. For example, after creating our proton, electron and photon

```
TElectron myElectron (aMomentum_K1);
TProton myProton (aMomentum_P1);
TPhoton myPhoton (aMomentum_Q);
```

we apply the methods `EmitPhoton()` to `myElectron` and `AbsorbPhoton()` to `myProton`:

```
myElectron.EmitPhoton(myPhoton);
myProton.AbsorbPhoton(myPhoton);
```

where we have assumed that both `TElectron` and `TProton` derive from `TChargedParticle`, and that the methods `EmitPhoton()` and `AbsorbPhoton()` in the class `TChargedParticle` are implemented to guarantee momentum conservation. This might be done as follows:

`myChargedParticle.EmitPhoton(myPhoton)` sets `myChargedParticle`'s momentum to

$$aMomentum_K2 = aMomentum_K1 - aMomentum_Q,$$

while `myChargedParticle.AbsorbPhoton(myPhoton)` sets `myChargedParticle`'s momentum to

$$aMomentum_K2 = aMomentum_K1 + aMomentum_Q.$$

where aMomentum_K1 is the initial value of `myChargedParticle`'s momentum. This representation of

the interaction of two objects mediated by a third one (messenger) is more closely related to that of QP shown in Fig. 3.

When describing the dynamics, it is convenient to use some objects, say from the class `TVector3D`, just as if they were built in types. For example, we might want to add and subtract `TVector3D` objects as in the momentum conservation equations of above. In the interface of the class `TVector3D` presented in [Appendix A](#) we have overloaded the assignment, the multiplication, and the addition/subtraction operators. *Overloading* an operator means that the compiler recognizes by context which operator implementation is being requested by the user in expressions such as the above ones. In other words, by overloading, for example, the `+` and `-` operators, we "teach the compiler" how to add/subtract `TVector3D` objects (using the symbols `+` and `-` in the familiar way). *Operator overloading* is a very nice and quite useful feature of some OO languages.

As the exchange energy between relativistic interacting particles increases other interaction processes become important. Consider an electron and its antiparticle, the positron. Given these objects at some initial time, at a later time they "vanish" and transform themselves into, say, a proton object and an antiproton object. This process is represented in the Feynman diagram of Fig. 4. This is an interesting example of objects completely losing their identity. Also, as in the scattering process of Fig. 3, this process is mediated by a photon object. This physics is not "naturally" captured in the current C++ language. [But, of course, it can be modelled]. In fact, OO languages other than C++ may offer more natural mechanisms for such a modelling, for example Smalltalk's "become" method.

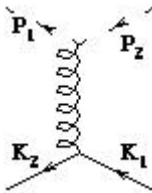


Figure 4. Feynman diagram showing electromagnetic electron-positron annihilation. Two different objects (electron and positron) in some initial states ($|K1\rangle$ and $|K2\rangle$) "annihilate" each other. An intermediate object is formed (photon), which then "becomes" two distinct objects (proton and anti-proton) in states $|P1\rangle$ and $|P2\rangle$.

As we have seen in this and the previous sections, in QP states have a "probabilistic" interpretation, and objects can sometimes confuse/lose their identities. They are summarized below:

- 1) When measuring a quantum system, state vectors just predict probabilities for objects to be in some states. Seldom does the state vector predict with certainty a unique state (this case corresponds to state vectors that are eigenstates of the complete set of observables).
- 2) If there are many identical quantum objects (i.e., objects that belong to the same class) confined to a "small" region in space their identity can get confused. This is a consequence of the uncertainty principle as position/momentum of objects cannot be measured simultaneously with infinite precision.
- 3) Finally the "worst case" occurs in relativistic quantum mechanics. In this case, objects can completely lose their identity. Then, objects of a given class could transform into objects of another (e.g., electron/positron into proton/antiproton). This effect is a consequence of both the uncertainty principle and also of the energy-mass equivalence;

i.e., the "stuff" encapsulated by the electron/positron objects transforms into an equivalent "stuff" now encapsulated by other types of objects: proton/antiproton.

In the OM literature usually states are not "probabilistic" and objects have a sharp identity as in "classical physics"[3]. On the other hand, this is not a limitation of the OM which is equally well suited to model QP. Design of appropriate classes and methods may be challenging.

VI. Object-Oriented Design And Effective Hamiltonians. Frameworks

Suppose we would like to design and implement an air traffic control system. How to identify the key abstractions in the problem space? How to identify classes and objects in an OO design? What are the possible relationships among classes? Classes (such as `TAirplanes`, `TRunways`,...) are static and their relationships are fixed before the execution of a program. In contrast, objects (such as a new runway, new airplane...) are dynamic. They are constantly created and destroyed during the lifetime of a typical application. The identification of appropriate classes and objects is the hardest problem of OO design[1-3] In fact, in this section we identify analogies between OO design and the problem of finding the "elementary components" of complex systems (for example, superconductors). In many-body theory, this is the problem of finding the effective Hamiltonian that describes the physics of interest.

At first glance, OO design may seem straightforward: just mimic the objects of the outside world and their interactions, relevant to the piece of reality to be captured in software. This is not the case in real OO design. In general, reality is too complex to be fully captured in a design. On the other hand, many of the outside objects/interactions may be totally or partially irrelevant to our problem. For example, in a naive design of a payroll one could include `TEmployee` objects, `TCheck` objects, `TPen` objects, `TInk` objects, `TBank` objects,.... On the other hand, a more sound design might just introduce `TEmployee` objects that pay themselves recognizing the irrelevance of other objects such as `aBank`, `anInk`, `aPen`, etc. to the problem of interest. In the process of abstraction, one "invents" a `TEmployee` object with properties which do not correspond to those of a "real" employee, but substantially simplifies the design. Moreover, to obtain tractable and sound designs, one may introduce "abstract", "intangible", or "totally imaginary" objects that "do not have any resemblance to reality". For example, in a "computer graphics window" design, one may use `TAnimation` objects (i.e., translation operators) that move things around, and objects of the types `TShape`, `TColor`, `TSize`, In the design of a complex system one has to identify key abstractions, propose some candidate classes and objects to mimic the system, and conjecture some class taxonomy. Some of this has been briefly discussed in Section IV.

What are observables? It depends on the observer's interest and ways of probing the system. For example, given some `TAirplane` objects in `anAirport`, one may be interested in either their size, color, when they were built, airlines to which they belong, the number of available seats, the type of engine, etc. Choosing the appropriate "degrees of freedom" usually is a highly non-trivial exercise, and OO designs depend critically on this choice. In QP one is confronted with similar situations, for example, given a proton, its momentum might be the observable of interest for some low energy experiments, at higher energies

one may be interested in the momenta of each quark. Alternatively, the proton's spin may be one of the selected observables if interested, say, in the hydrogen hyperfine structure. Or protons' degree of freedom may be totally irrelevant, say, if describing hydrodynamic properties of water; even though protons are "part of" water.

How does one identify "classes and objects" and their relationships and interactions in Natural Sciences? The components of a system are usually determined by how the system "breaks" under "minimal forces". For example, oxygen gas is made of oxygen molecules. As we apply stronger forces (at higher collision energy), molecules break down, and now the system is made of oxygen atoms, ions and free electrons. Substantially increasing the energy of our probes, we see our system composed of electrons, protons and neutrons. At even higher collision energy one discovers that protons and neutrons are made of quarks. This, again, is an illustration of the hierarchical structure of Nature. "Minimal unit" is an ambiguous concept that strongly depends on "context": depth of penetration into the structure, physical conditions and phenomena to be described (i.e., the kind of experiment selected to explore the system). Given our oxygen gas at room temperature, we view it as composed of molecules when interested in specific heat. On the other hand, if interested in, say, electrical conductivity we instead regard the gas as made of positive and negative ions.

One of the most basic problems of Physics is to "find" the "fundamental objects" or "elementary components" of complex systems. This problem is illustrated by superconductivity. Superconductors are materials that offer almost no electrical resistance. Hence electrical currents circulate for months with very little heat dissipation. Conduction properties of metals can be explained in terms of interacting ion and electron objects, but interactions among these type of objects cannot help explaining superconductivity. Superconductivity was discovered in 1911, but it took almost 50 years find a satisfactory theory. What are the relevant "objects" to describe superconductivity? The BCS (Bardeen, Cooper and Schriffer) theory of superconductivity has shown that "the" sought "object" is the so-called Cooper pair. A Cooper pair is an "atom" made of two electrons that weakly attract each other (attraction mediated by the host ions) and which behaves like a whole. This is an important point, although "free" electrons repel each other (Cooper pairs cannot exist in isolation from the material environment), in the appropriate environment (inside a superconducting metal) they, indeed, form bound states and behave as a whole identity. Cooper pairs are called quasiparticles, as they do not exist in isolation. Quasiparticles, as animals, need their own habitats to survive (e.g., superconductivity only occurs for a few metals in some range of temperature, external fields,...). In contrast, protons and electrons are particles. They can exist in freedom. The Cooper pairs (bosons) are the carriers of electricity in a superconductor, and are much more effective in their task (conducting electricity) than electrons (fermions), carriers of electricity in normal metals.

How do we "solve" a complex many-body problem in physics? Let us again discuss superconductivity. Given a metal made of ions and electrons, one searches for the appropriate class or classes that capture the physics of the problem. As reality is too complex, only essential features can be incorporated if the model is to be mathematically (and/or computationally) tractable. One hopes to write the so-called effective Hamiltonian H_{eff} of the system

$$H_{\text{eff}} = H_0 + H_I ,$$

with H_0 the free Hamiltonian containing the fields (classes) relevant to our problem. For example, in the superconductivity case, the class of interest is $T_{\text{CooperPair}}$ although naively, one could have said that the "right" classes were T_{Ion} and T_{Electron} . The effective Hamiltonian is an operator, written in terms of the $T_{\text{CooperPair}}$ creation and annihilation operators. H_0 describes a system of "non-interacting" Cooper pairs (that is, there may be

