ECONSTOR
Make Your Publications Visible.

A Service of
ZBW
Leibniz-Informationszentrum Wirtschaft
Leibniz Information Centre for Economics

Virius, Miroslav

Working Paper

Object Oriented Computing

Papers, No. 2004,19

Provided in Cooperation with:
CASE - Center for Applied Statistics and Economics, Humboldt University Berlin

Suggested Citation: Virius, Miroslav (2004) : Object Oriented Computing, Papers, No. 2004,19, Humboldt-Universität zu Berlin, Center for Applied Statistics and Economics (CASE), Berlin

This Version is available at:
https://hdl.handle.net/10419/22193

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.

WWW.ECONSTOR.EU

Mitglied der Leibniz-Gemeinschaft

Object Oriented Computing

Miroslav Virius

Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering,
Czech Republic
virius@km1.fjfi.cvut.cz

In this contribution, the basic overview of the Object Oriented Programming and its usage in computation is given. Concepts of class, encapsulation, inheritance, and polymorphism are introduced. Some additional concepts like interface and Design Patterns are briefly discussed. Schematic examples in C++ are given.

1 Introduction

Object Oriented Programming (OOP) is a preferred methodology in contemporary software development. OOP may be considered as a continuation of the well known ideas of Structured Programming and Modular Programming. If properly used, it leads to well structured code which is easy to debug and easy to maintain.

1.1 First Approach to Objects

Every computer program may be considered as a software model of a real problem. It follows, that two basic domains should be taken into account during the analysis of the problem and design of the program: the *problem* domain, which is part of the real world, and the *model* domain, which is a mapping of the problem domain to the computer program.

The problem domain consists of a set of interacting objects. Selected objects of the problem domain must of course correspond to data structures in model domain and the interactions of objects in the problem domain must correspond to the operations with these data structures. That is, the interactions of objects in the problem domain will be represented by procedures and functions dealing with these data structures.

Example. Consider modelling the interactions of elementary particles in a detector using the Monte Carlo method. (Design and analysis of Monte Carlo experiments is discussed in depth in Chapter **??**.) The problem domain of this experiment simulation consists of the detector, the particle source, the air surrounding the experimental apparatus, of many elementary particles and, of course, of a statistical file containing the simulation results. It follows, that the model of the experiment should contain a suitable representation of the detector, a suitable representation of the particle source, statistical file, etc.

The representation of the elementary particle source may consist of the data representing its coordinates in a given coordinate system, of a description of the spectrum of the source (i.e. of probability distributions describing the emission of various types of particles, their direction, energy and other characteristics of emitted particles) etc.

1.2 Note on Unified Modelling Language

To formalize object-oriented analysis and design, the Unified Modelling Language (UML) is widely used. UML consists of a set of diagrams that describe various aspects of the problem solved. We use some UML diagrams in this chapter. A short introduction to the UML is given in Sec. 3; full description of the UML may be found in Booch (1999).

2 Objects and Encapsulation

In the model domain, the term *object* denotes the data representation of the objects of the problem domain, together with the operations defined on this data.

This means that we define a data structure together with the operations with it. These operations are usually called *methods*. The object's data are denoted as *attributes*; the data and methods together are denoted as *members* of the object.

A basic rule of OOP requires that the methods should be used for all the manipulations with object's data. Methods of the object are allowed to access the data; no other access is permitted. (We shall see that under some circumstances it is acceptable to violate this rule.) This principle is called *encapsulation* and is sometimes presented by the "wall of code around each piece of data" metaphor.

Note. Methods that return the value of the attribute (data member) `X` usually have the identifier `GetX()`; methods that set the value of the attribute `X` usually have the identifier `SetX()`. In some programming environments, these identifiers may be required. These methods are called *getters* and *setters*,, respectively.

2.1 Benefits of Encapsulation

So far, there is nothing new in encapsulation: This is implementation hiding, well known from modular programming. The object may be considered as a module and the set of the methods as its interface.

The main benefit of encapsulation is that the programmer may change the implementation of the object without affecting the whole program, if he or she preserves the interface of the object. Any change of the data representation will affect only the implementation of the methods.

Example. Let's continue with the Monte Carlo simulation of the experiment with elementary particles. The object representing the detector will, of course, contain the coordinates of some important points of the detector. The first idea could be to use Cartesian coordinates; in later stage of the program development, it will be found that the spherical coordinates will suit better — e.g., because of the detector shape and program performance.

If it were allowed to manipulate the detector data directly by any part of the program, all the parts of the program that use this data should be changed. But if the encapsulation is properly applied and the data is manipulated only by the methods of the detector, all that has to be changed is the implementation of some detector methods.

2.2 Objects and Messages

OOP program is considered as the program consisting only of objects that collaborate by means of the *messages*. This may seem a little strange, but in this context, *to send a message to an object* means *to call a method of this object*. A message means a request for an operation on the object's data, i.e., a request to perform a method.

The object may receive only those messages for which it has corresponding methods. Sending a message that the object does not recognize causes an error. It depends on the programming language whether this error is detected in the compile time or in the run time. (In C++, it is detected in the compile time.)

2.3 Class

Objects of problem domain may often be grouped into *classes*; one class contains objects that differ only in the values of some properties. The same holds for the objects in the model domain. The classes of objects in the problem domain are represented by user-defined data types in OOP programs called *object types* or *classes* .

The term *instance* is used to denote a variable, constant, or parameter of an object type. It is equivalent to the term *object*.

Class Members

Up to now, we have considered the class as a data type only; it serves as a template for the creation of instances. But in OOP, the class may have its own data and its own methods and may receive messages.

Data members that are part of the whole class (not of particular instances) are called *class data members* or *class attributes* and the methods that correspond to messages sent to the whole class are called *class methods*. Non-class members, attributes, as well as methods are, if necessary, denoted *instance members*.

Class data members contain data shared among all the instances of the class; class methods operate on class attributes. (From the non-OOP point of view, class data members are global variables hidden in the class, and class methods are global functions or procedures hidden in the class.)

Note. Class data members are often called *static data members* and class methods are called *static methods* in C++, Java, and some other programming languages, because they are declared using the `static` keyword in these languages.

Note. The class in C++, Java and many other OOP languages may contain definitions of other types, including other classes, as class members. Even though the so called nested classes are sometimes very useful, we will not discuss them in this article.

Note. The class in the OOP may be considered as an instance of another class; this leads to the concept of *metaclass*. Metaclass is a class that has only one instance — a class. You can find metaclasses in pure OOP languages like Smalltalk. We will not discuss the concept of metaclass here.

Example. We may suppose — at some level of abstraction — that the representation of all the particles in the Monte Carlo simulation of the experiment with the particles is essentially the same. Thus, every individual particle belongs to the *class of particles*. It follows that the model will contain the `Particle` class, and the program will contain the definition of the corresponding data type (and of course some instances of this type).

Because every particle has its own mass and velocity, the `Particle` class will contain the declaration of four data items representing particle mass and three components of the particle velocity vector. The `Particle` class should also contain methods to set and to get the values of these data items. (Later on, we will see that even other methods are necessary — e.g., a method for the interaction with the detector.)

It is also necessary to know the total number of generated particles and the actual number of existing particles in the simulation program. These numbers of the particles do not describe an individual particle and so they cannot be data members of any `Particle` instance; it is the task of the whole `Particle` class to hold these data. So they will be stored in the class attributes (because

we use the C++, we may say in static attributes) of type `int`, and they will be accessed by class methods (static methods).

Definition of the `Particle` class in C++ will be as follows:

```
// Particle class definition in C++, first approach
class Particle
{
public:
   // Constructor
   Particle(double _mass, double vX,
        double vY, double vZ);
   // Instance methods
   ~Particle() { --actual; }              // Destructor
   double GetMass() { return mass; }
   void SetMass(double m){ mass = m; }
   void SetVelocity(double vX, double vY, double vZ);
   double GetVelocityX() { return velocityX; }
   // Performs the interaction with the detector
   virtual void Interact(Detector *aDetector);
   // ... and other methods
   // Class methods
   static int GetActual() { return actual; }
   static int GetTotal() {}
private:
   // Instance data members
   double mass;
   double velocityX, velocityY, velocityZ;
   // Class data members
   static int actual;
   static int total;
};                   // End of the class declaration

// Definition of the static attributes
int Particle::actual = 0;
int Particle::total = 0;

// Definition of the constructor
Particle::Particle(double _mass, double vX,
                   double vY, double vZ)
: mass(_mass), velocityX(vX), velocityY(vY), velocityZ(vZ)
{
 ++actual; ++total;
}
// And other method definitions
```

We will not discuss the syntactic rules of the class declaration in the C++ here — this can be found in any textbook of this programming language, e.g., in Stroustrup (1998). We only note a few points.

This class contains the instance attributes `mass`, `velocityX`, `velocityY`, and `velocityZ`, and the class attributes `actual` and `total` (note the `static` keyword in their declarations). It follows that every instance of the `Particle` class will have its own data members `mass`, `velocityX`, etc. On the other hand, no instance will contain the data members `total` or `actual`. These are global variables shared by all instances and they exist even before the first instance of the `Particle` class is created and after the last one is destroyed.

The `Particle()` method is a special method called *constructor* and it serves the construction of new instances. It is invoked as a response to the message requesting the creation of a new instance of the class. (Even though it is a class method, its declaration in C++ does not contain the `static` keyword.) Its task is to initialize instance attributes. In our example, it also actualizes the values of the two class attributes.

The `~Particle()` method is another special method called *destructor* that prepares the instance for decay. In our example, it decreases the number of existing particles, because the instance for which the destructor is called will be immediately destroyed. (This is — unlike the constructor — the instance method. Note, that in garbage collected OOP languages, e.g., in Java, destructors are not used.)

2.4 Object Composition

One object in a program may exploit the services of another object. It may call the methods of any other independent object, or it may contain another object as a data member. The second approach is usually called *object composition*, even though it is typically implemented as composition of the classes.

Note. An object may not contain another object of the same class, of any class containing an object of the same class or of any derived class as data member. It may, of course, contain the pointers or the references to objects of any of these classes.

Example. Consider the particle source in the Monte Carlo simulation. It will be an instance of the `Source` class. For the simulation of the random processes of the emission of a particle, we will need a random number generator. The random number generator will be implemented in the program as an instance of the `Generator` class and will be based on the theory discussed in Chapter **??**. (The `Generator` class is an example of a class that has been found during the design of the `Source` class. It does not appear in the original formulation of the problem.)

This means that the `Source` class will contain an instance of the `Generator` class or a pointer to an instance of that class:

```
class Source
{
public:
   Source();
   Particle* Generate(); // Returns pointer to new particle
   // ... and other methods
private:
   Generator *gen;
   // ... and other private members
};
```

2.5 Access Control

Note the `private` and `public` *access specifiers* in the class declarations above. The `public` specifier declares that all the subsequent members of the class are public, i.e., they are accessible from any part of the program. The public members of the class constitute the *class interface*. The class interface usually contains only some methods and constant attributes. (Constant attributes may be accessed directly. This does not violate the encapsulation, because constant attributes cannot be changed.)

The `private` specifier means that the following members of the class are private, i.e., accessible only from the methods of the class. In other words, private members are *implementation details* of the class that can not be used by other parts of the program. Changes of private parts of the class do not change the class interface and do not affect other parts of the program.

Later on, we will see the third access specifier, `protected`. Protected members are accessible only from the methods of this class and from all the derived classes. So, they constitute the *class interface for derivation*, that may be wider than the interface of the class for common use. We will discuss the inheritance in Section 4.

The access specifiers help to implement the encapsulation. Note that in C++, as well as in many other object oriented languages, the subject of access control is the class, not the individual objects (instances). So any method called for an instance of the given class may use all private members of another instance of the same class.

3 Short Introduction to the UML

In Sec. 1.2 we have mentioned the UML. This is a modelling language based on a set of diagrams describing various aspects of the problem solved:

- The *class diagram* describes the classes used in the problem and their mutual dependencies.
- The *object diagram* describes all objects (instances) in the problem and their mutual dependencies.

- The *activity diagram* describes activities of the objects.
- The *state diagram* describes the states of objects and their possible changes and transitions.
- etc.

We will use only class diagrams in this article.

The class in the class diagram is presented as a rectangle containing the name of the class. It usually contains also the names of the methods and the names of the attributes; both may be prefixed by symbols representing their access specification (the `+` sign for public members, the `-` sign for private members and the `#` for protected ones). The name, the attributes and the methods are in the class icon separated by horizontal lines. If not necessary, attributes and methods may be omitted. Fig. 1 shows the icon of the `Source` class as we have designed it in Section 2.4.

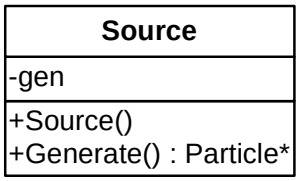

Fig. 1. The full UML icon of the `Source` class.

Associations (i.e., any relations) among classes in UML class diagrams are represented by lines connecting the class icons ended by arrows; as a description, the multiplicity of the relation may be given. For example, the number appended to the line connecting the `Source` and the `Particle` classes in Fig. 2 express the fact that one particle source may emit any number of particles. The number appended to the line connecting the `Source` and the `Generator` class express the fact that one source uses only one random number generator.

Object composition is expressed by the arrow ending with a filled diamond. Fig. 2 shows relations among the `Source`, `Particle`, and `Generator` classes. Simplified class icons are used.

4 Inheritance

Inheritance is a very powerful tool used in OOP to derive new classes from existing ones. First, look at an example.

Example. Investigating our Monte Carlo simulation more deeply, we find, that various types of elementary particles can be involved: photons, neutrons, neutrinos, etc.

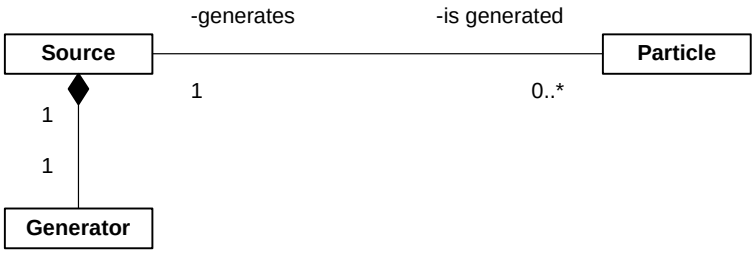

Fig. 2. Relations among the `Source`, `Particle` and `Generator` classes.

On the one hand, we may conclude that one common data type, the `Particle` class, is sufficient for the representation of all the different particles, because they have many common features:

- Every particle has a velocity vector,
- every particle has a mass, a spin, and electrical charge,
- every particle has its halftime of decay,
- every particle may interact with the detector,
- etc.

On the other hand, the way of the interaction with the detector is substantially different for different types of the particles. In some cases, it is described by mathematical formulae, in other cases it is described by measured data only. It follows that the operation representing the interaction of the particle in the detector must be implemented in a different way for different types of the particles, and this leads to the conclusion that different types of simulated particles have to be represented by different object types in the program; but these types share many common properties.

This situation — closely related, but different classes — can be expressed in the program model: OOP offers the mechanism of *inheritance*, which is the way of deriving one class from some other one (or other ones).

The class a new type is derived from is usually called the *base class*.

4.1 Base class and derived class

The derived class *inherits* all the public and protected members of its base class or classes. This means that the derived class contains these members and may access them without any constrains. Private members are not inherited. They are not directly accessible in the derived class; they may be accessed only by the access methods inherited from the base class.

The derived class may add its own data members and methods to the inherited ones. The derived class may also redefine (*override*) some of the methods defined in the base class. (To override a method in a derived class means to implement a different response to the same message in the derived

class.) In this case, the signature, i.e., the identifier, the return type, the number, and the types of the parameters of the overriding method in the derived class should be the same as the signature of the overridden method in the base class.

No members of the base class may be deleted in the inheritance process.

The set of all the classes connected by inheritance is usually called the *class hierarchy*.

Note that in some programming languages there are exceptions to the above rule. For example, the constructors, destructors, and overloaded assignment operators are not inherited in C++. Instead, the constructor of the derived class always calls the base class constructor and the destructor of the derived class always calls the base class destructor. The same holds for the default assignment operator of the derived class. Of course, this may be considered as a generalized form of inheritance.

4.2 Generalization and Specialization

The base class always represents a concept that is more general and more abstract, than the concept represented by the derived class; it follows that the derived class represents a more specialized concept than the base class. In other words, the derived class always represents a *subclass* — or a subtype — of its base class. *Any instance of the derived class is also considered to be an instance of the base class.*

The interface of the base class is a subset of the interface of the derived class.

Consequently, an instance of the derived class may be used everywhere where an instance of the base class is expected. This rule may significantly simplify the operation with instances of many similar classes.

In the UML class diagram, the inheritance is represented by an arrow ending with triangle (not filled). The arrow leads from the derived class to the base class.

Example. Consider the `Particle` class in our Monte Carlo simulation. This is a general concept that can be used to describe common features of all the particles involved. But in the simulation, concrete types of particles — e.g., protons, electrons, etc. — will be used.

Consequently, we will use the `Particle` class as the base class of the particles hierarchy that will contain the common data members and the common methods of all the particles. All the classes representing concrete particle types will be derived from the `Particle` class — see Fig 3.

We will show here only the declaration of the `Electron` class.

```
class Electron: public Particle
{
public:
   Electron();
```

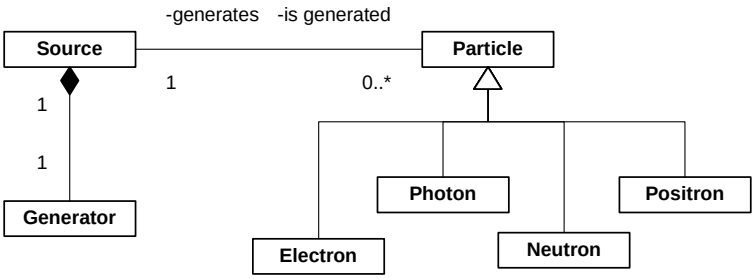

Fig. 3. The `Particle` class as a base class of concrete particle types.

```
   void SetCharge(double _charge) { charge = _charge; }
   virtual void Interact(Detector *aDetector);
private:
   double charge;
};
```

The declaration of the `Electron` class contains only the declaration of the constructor, two access methods and one data member. Nevertheless, the methods `GetvelocityX()`, `SetVelocityX()`, `GetActual()` etc., inherited from the base class, may be called for any instance of this class. This class changes — overrides — the implementation of the `Interact()` method.

On the other hand, data members `mass`, `velocityX`, `actual`, etc. are not directly accessible in the `Electron` class. These data members are in the base class declared as private and the derived class must manipulate them only using the public access methods. So the following fragment of the definition of the `Electron::Interact()` method is incorrect:

```
// Error: velocityX, velocityY and velocityZ
// are inaccessible in the Electron class.
void Electron::Interact(Detector *aDetector)
{
   double velocity = sqrt(velocityX*velocityX +
             velocityY*velocityY + velocityZ*velocityZ);
   // ... and so on ...
}
```

The correct form uses the access methods inherited from the base class:

```
// Correct form of the previous code fragment
void Electron::Interact(Detector *aDetector)
{
   double velocity = sqrt(GetVelocityX()*GetVelocityX() +
             GetVelocityY()*GetVelocityY() +
             GetVelocityZ()*GetVelocityZ());
```

```
   // ... and so on ...
}
```

Of course, this is sometimes inconvenient. If we change the access specifiers of these data members in the base class to `protected`,

```
// Particle class definition revised
class Particle
{
public:
   // Public members as before
protected:
   // Instance data members
   double mass;
   double velocityX, velocityY, velocityZ;
   // Class data members
   static int actual;
   static int total;
};
```

the problems with access to data members will not appear. On the other hand, this violates the encapsulation of the base class and it may have a negative impact on the clarity and maintainability of the program.

4.3 Using Base Class as Common Interface

As stated above, instances of derived classes may be used anywhere instances of the base class are expected. This gives us very powerful tool to deal with the objects of the classes derived from the same base class in a uniform manner.

Example. In the Monte Carlo simulation of the particle experiment, we may first store all the emitted particles in a suitable container, exclude particles, that do not hit the detector etc., and after that preprocessing, let the remaining particles interact with the detector. Consider the following fragment of code:

```
const int N = 1000000;     // Number of particles to emit
vector<Particle*> store;   // Store for particles
Source sce1;               // Particle source
Detector det;              // Detector in the experiment
for(int i = 0; i < N; i++)
   store.push_back(sce1.Generate());
// ... some preprocessing of the set of the particles
for(int i = 0; i < store.size(); i++)
   store[i] -> Interact(det);
```

The `store` variable is a vector (dynamically sized array) of pointers to `Particle`, the base class of all the elementary particles involved. This allows

us to store pointers to instances of any class derived from the `Particle` class in this container.

The expression

```
store[i] -> Interact(det);
```

represents the call of the `Interact()` method of the particle pointed to by `store[i]`. (In fact, this statement calls the method of the `Particle` class. To ensure that the method of the actual class of the particle is called, the method needs to be declared with the `virtual` specifier. This will be discussed later in Section 5, *Polymorphism.*)

This example demonstrates that the base class defines the common interface for all the derived classes.

Technical Note

The conversion from the derived class to the base class is automatic, but in some situations deserve special attention. Consider the following assignment:

```
Electron e;                    // Non-dynamical instances
Particle p;
p = e;
```

After this statement has been executed, the `p` variable will contain an instance of the `Particle` class, not an instance of the `Electron` class! The reason is simple: The declaration

```
Particle p;
```

reserves store only for the `Particle` instance, so there is no place for the additional data members declared in the `Electron` class. The only way how to execute the assignment is to convert the derived class instance `e` to the base class first.

The same problem arises in the case of passing function arguments by value. Having the function

```
void process(Particle p);      // Pass by value
```

it is possible to write

```
process(e);                    // e is Electron
```

but the instance `e` of type `Electron` will be first cast (converted) to the `Particle` type. This cast leads to the loss of information.

These problems may be avoided if we use dynamical instances only. If we rewrite the preceding declarations into the form

```
Electron *ep = new Electron;   // Dynamical instances
Particle *pp;
pp = ep;                       // OK
```

the `pp` variable will still contain the pointer to the instance of the `Electron` class. (The type of the pointer is converted, not the type of the instance.)

The parameter of the `process()` function should be passed by the pointer or by the reference. In both cases, the original instance is accessible in the function body and no information is lost.

In Java, C# and other OOP languages, that use dynamical instances of the object types only and manipulate them by references, these problems do not appear.

4.4 Inheritance, or Composition?

In some cases, it is not clear, whether a new class should be derived from some suitable class by inheritance or whether object composition should be used.

There are two questions that should be answered in this case:

- *Is* the new class a special case of the base class proposed?
- *Has* the new class a data member of the proposed class?

This is known as the *IS A – HAS A* test. Only if the answer to the first question is *yes*, the inheritance may be considered, otherwise the composition should be used.

Example. Consider the `Source` class, representing the source of elementary particles in the Monte Carlo simulation. It will be based on some generator of random numbers represented in our program by the `Generator` class. In other words, the `Source` seems to be the `Generator` class with some added functionality. Should we derive the `Source` class from the `Generator` class?

If we apply the IS A – HAS A test, we find that the particle source is not a special case — a subclass — of the random number generator. It uses the random number generator, so it may contain it as a data member, however, the inheritance should be avoided in this case.

Consider for an instant that we use the inheritance to derive the `Source` class from the `Generator` class,

```
// Wrong use of the inheritance
class Source: public Generator
{
   // Body of the class
}
```

This would mean that we can use a `Source` instance everywhere the `Generator` instance is expected. But in the Monte Carlo simulation, the random number generator is necessary even in other parts of the program, e.g. for the determination of the interaction type, for the determination of the features of the secondary particles resulting from the interaction (if any) etc. However, in these parts of the program, the `Source` class may not serve as the random number generator.

Such a design of the `Source` class may cause that some typing errors in the program will not be properly detected and some mysterious error messages during the compilation will be reported; or even worse — it may lead to runtime errors hard to discover.

4.5 Multiple Inheritance

The class may have more than one base class; this is called *multiple inheritance*. The class derived from multiple base classes is considered to be the subclass if all its base classes.

Multiple inheritance may be used as a means of the class composition.

This feature is supported only in a few programming languages — e.g., in C++ (see Stroustrup, 1998) or in Eiffel (see Meyer, 1988). In Java, C#, Object Pascal and some other languages it is not supported. Multiple inheritance poses special problems, that will not be discussed here.

5 Polymorphism

At the end of the previous section, we have seen that instances of many different classes were dealt with in the same way. We did not know the exact type of the instances stored in the `store` container; it was sufficient that they were instances of any class derived from the `Particle` class.

This feature of the OOP is called *polymorphism*. Polymorphism means that instances of various classes may be used in the same way — they accept the same messages, so their methods may be called without any regard to the exact type of the instance. In some programming languages (e.g., in Java), this is automatic behavior of the objects (or of their methods), in some programming languages (e.g. in C++) this behavior must be explicitly declared.

There are at least two ways to achieve polymorphic behavior: The use of the inheritance and the use of the interfaces. The interfaces will be discussed in 5.4.

Example. Let's consider once again the example given at the end of the *Inheritance* section. The expression `store[i]` is of type "pointer to the `Particle` class", even though it in fact points to an instance of the `Electron`, `Photon`, or some other derived class.

It follows that the statement

```
store[i] -> Interact(det);     // Which method is called?
```

might be interpreted as the call of the `Particle::Interact()` method, even though it should be interpreted as the call of the `Interact()` method of some derived class.

5.1 Early and Late Binding

The previous example shows that there are two possible approaches to the resolution of the type of the instance for which the method is called, if the pointer (or reference) to the instance is used:

- *Early binding.* The type of the instance is determined in the compile time. It follows that the static (declared) type of the pointer or reference is used. This is the default for all methods in C++, C#, or Object Pascal.
- *Late binding.* The type of the instance is determined in the run time. It follows that the actual type of the instance is used and the method of this type is called. This is always used for the methods in Java. In C++, the `virtual` keyword denotes the methods using the late binding.

Late binding gives the class polymorphic behavior. On the other hand, late binding is less effective than early binding, even though the difference may be negligible. (In C++ on PCs, the difference between the late and the early binding is usually one machine instruction per method call.)

Any method that might be overridden in any of the derived classes should use the late binding.

Note. In C++ and other OOP languages in which the late binding must be declared, the classes containing at least one virtual method are called *polymorphic classes*. Classes without any virtual method are called *non-polymorphic classes*. In languages like Java, where all the methods use late binding by default, all the classes are polymorphic.

5.2 Implementation of the Late Binding

In this subsection, some low level concepts will be discussed. They are not necessary for understanding the basic concepts of the OOP, but they can give better insight in it.

We will explore one the common way of implementation of the late binding, i.e., of the determination of the actual type of the instance for which the method is called.

This is based on the so called *virtual method tables*. The virtual method table (VMT) is the hidden class data member that is part of any polymorphic class. Any polymorphic class contains exactly one VMT. (The hidden class member is a class member the programmer does not declare — the compiler adds it automatically.)

The VMT is an array containing pointers to all the virtual methods of the class. Any derived class has its own VMT that contains pointers to all the virtual methods (even those that are not overridden in this class). The pointers to the virtual methods in the VMT of the derived class are in the same order as the pointers to corresponding methods in the base class.

Any instance of the polymorphic class contains another hidden data member — the pointer to the VMT. This data member is stored in all the instances at the same place — e.g. at the beginning.

The method call is performed in the following way:

1. The program takes the instance, for which the method is called.
2. In the instance, it finds the pointer to the VMT.
3. In the VMT, it finds the pointer to the method called. In all the VMTs this pointer is in the same entry; e.g., the pointer to the `Interact()` method might be in the VMT of the `Particle` class and in the VMTs of all the classes derived from the `Particle` in the first entry.
4. The program uses this pointer to call the method of the actual class of the instance.

Fig. 4 illustrates this process for the `Particle` base class and two derived classes. The values stored in the VMT are set usually at the start of the program or when the class is loaded to the memory. The values of the pointer to the VMT in the instances are set automatically by the constructor.

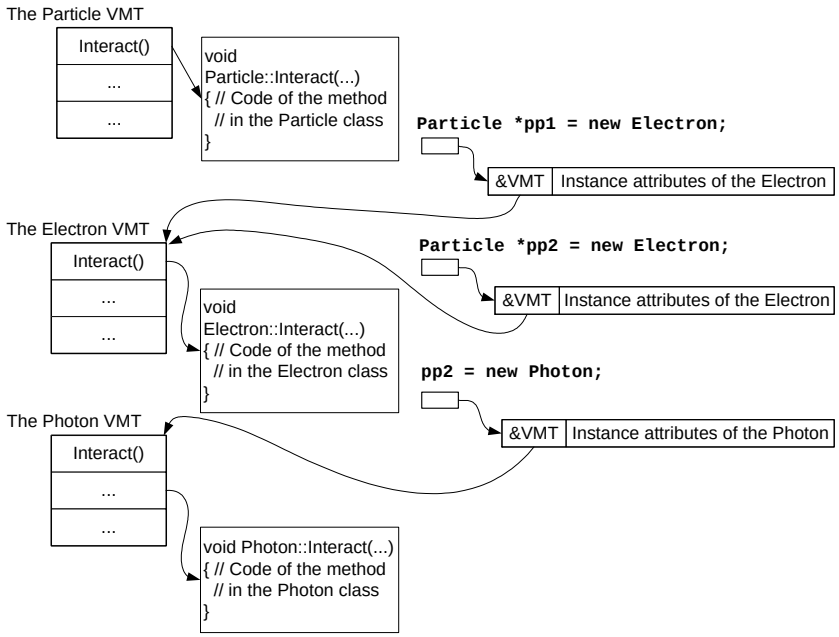

Fig. 4. Typical implementation of the late binding.

5.3 Abstract Class

In some cases, the base class represents such an abstract concept that some operations with instances of this class cannot be implemented. Nevertheless, at least the stub of the corresponding method should be present in the class, because this class is used as a base class and determines the common interface for all the derived classes.

In OOP such a class is called the *abstract class* and such an operation is called the *abstract method*. Abstract methods have no implementation (no method body).

It is not allowed to create instances of the abstract classes and it is not allowed to call the abstract methods. It is of course possible to define pointers or references to abstract classes.

The abstract classes serve as base classes. If the derived class does not implement any of the inherited abstract methods, it will be abstract like the base class. The abstract class

- defines the interface of the derived classes,
- provides the implementation of non-polymorphic methods for the derived classes, and
- offers a default implementation of non-abstract polymorphic (virtual) methods.

Note that the abstract classes are italicized in the UML class diagrams — see e.g. the `Particle` class in Fig. 5.

Example. Consider the `Particle` class defined above. How could the `Interact()` method be implemented?

For the derived classes, the situation is clear: If it is, e.g., the `Photon`, the interaction could be the photoelectric effect, the Compton scattering, or some other interaction known to particle physicists; probabilities of these phenomena are determined according to their total effective cross sections. For the other derived classes, there are other well defined possibilities that can be expressed in the program code.

However, there is no general interaction, that could be used to implement the `Interact()` method of the general `Particle`. On the other hand, this method must be declared in the `Particle` class as the part of the common interface of derived classes. If we omit it, the statement

```
store[i] -> Interact(det);
```

will not compile, because `store[i]` is the pointer to the `Particle` class that does not contain such a method.

So, the `Interact()` method should be declared as abstract (in C++, the abstract methods are called *pure virtual methods*). In the following revision of the `Particle` class, we omit all other parts of that class that are unchanged.

```
// Particle as an abstract class
class Particle
{
public:
   // Pure virtual method
   virtual void Interact(Detector *aDetector) = 0;
   // All other public members as before
protected:
   // All data members as before
};
```

5.4 Interfaces

The *interface* may be defined as a named set of methods and constants. This set may be empty.

The interface represents a way to achieve the polymorphic behavior; it is an alternative to inheritance. This concept is relatively new in OOP; it was first widely used in the Java language.

In languages that support interfaces, any class may declare that it *implements* the given interface. This means that the class will supply the implementations (bodies) of the methods in the interface.

Note the terminological difference: Even though interfaces are syntactically similar to the classes that contain only public abstract methods, they are not *inherited*, but they are *implemented*. In programming languages, that support interfaces, any class may implement many interfaces, even if the language does not support multiple inheritance.

The interface represents the type. If class `C` implements interfaces `I1` and `I2`, any instance of this class is an instance of type `C` and also an instance of type `I1` and of type `I2`.

The interface is usually represented by a small circle connected to the class icon in the UML class diagrams (see Fig. 5). It may also be represented by a class-like icon marked by the ≪`interface`≫ label ("stereotype"). Implementation of the interface is represented by a dashed arrow pointing to the implementing class (see Fig. 7).

5.5 Interfaces in C++

As we have chosen C++ as the language of examples, it is necessary to cover briefly the interfaces in this language. C++ does not support interfaces directly; nevertheless, interfaces may be fully simulated by abstract classes that contain only public abstract (pure virtual) methods, and the interface implementation may be substituted by the inheritance. This will be demonstrated by the following example.

Example. The Monte Carlo simulation may be time-consuming and it would be convenient to have the possibility to store the status of the simulation into a file, so that the computation might be interrupted and continued later.

It is clear that all the generated but not yet processed particles should be stored. The status of the particle source, and consequently the status of the random number generator, should be stored, too. This is necessary especially for debugging, because it ensures that we could get the same sequence of random number in repeated runs of the program, even if they are interrupted.

It follows that we have at least two different object types belonging to different class hierarchies that have a common feature — they will be stored in a file and later will be restored into their original state. It follows that all the classes involved should have suitable methods, e.g., `store()` and `restore()`.

The simulated experiment is represented by the `Experiment` class in the program and to store the experiment status is the task of this class; so we would like to implement in this class the `storeObject()` method to store objects passed as arguments. It follows that all the parameters — all the objects stored — should be of the same type.

The solution of this dilemma — the method requires objects of the same type as parameters, but we have objects of at least two distinct types belonging to different class hierarchies — is to use a suitable interface that contains the `store()` and `restore()` methods. We will use the `Storable` identifier for this interface. The `Source`, `Generator` and `Particle` classes should be modified as follows:

```
// Interface simulation in C++
class Storable
{
public:
   virtual void store(ostream&) = 0;
   virtual void restore(istream&) = 0;
};

class Generator: public Storable   // Interface implementation
{
public:
   virtual void store(ostream& out)
   {/* Store the generator */}
   virtual void restore(istream& in)
   {/* Read the generator and reconstruct it */}
   // ... Other methods and attributes as before
};

class Source: public Storable       // Interface implementation
{
public:
```

```
   virtual void store(ostream& out)
   {/* Store the source */}
   virtual void restore(istream& in)
   {/* Read the source from the file and reconstruct it*/}
   // ... Other methods and attributes as before
};

class Particle: public Storable  // Interface implementation
{
public:
   virtual void store(ostream& out)
   {/* Store the particle */}
   virtual void restore(istream& in)
   {/* Read the particle from the file and reconstruct it */}
   // ... Other methods and attributes as before
};
```

(`ostream` and `istream` are base classes for output and input data streams in the standard C++ library.) Fig. 5 shows the revised UML class diagram of these classes.

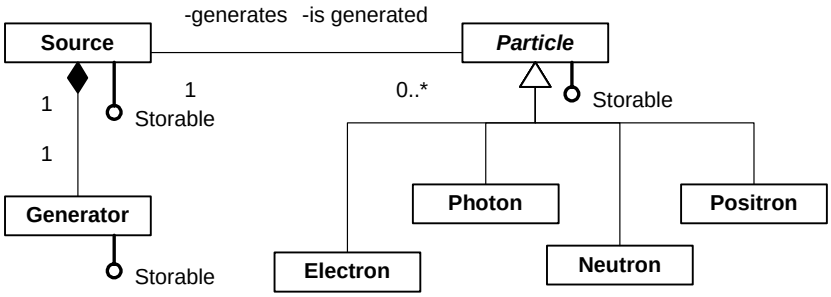

Fig. 5. The classes implementing the `Storable` interface.

Note that the `Particle` class is abstract, so it need not override the methods of the `Storable` interface. The classes representing the concrete particle types, `Electron` etc., inherit the implementation of the `Storable` interface; thus it is not necessary to declare this fact. Of course, if a derived class is not abstract, it must override the methods of this interface.

Implementation of the `Storable` interface allows us to define the method `Experiment::storeObject()` as follows:

```
void Experiment::storeObject(Storable& obj, ostream& out) {
    obj.store(out)
}
```

`Storable` interface serves as the common type of all the storable objects — particles as well as random number generators — in the program. This gives us the possibility to treat all these objects in our program in a uniform way.

6 More about Inheritance

Inheritance can be easily misused and this is often counterproductive. Poorly designed inheritance hierarchies lead to programs that are difficult to understand, contain hard-to-find errors, and are difficult to maintain. In this section, we give some typical examples.

6.1 Substitution Principle

In Section 4, *Inheritance*, we have seen that any instance of any derived class may be used where an instance of the base class is expected. This is sometimes called *the substitution principle*.

As far as we have seen, this is a syntactic rule: If you follow it, the program compiles. But we already know that for reasonable use of the inheritance, the derived class must be a specialization of the base class. Let's investigate more deeply, what it means.

"Technical" Inheritance

This problem is similar to the problem we have seen in Section 4.4. We have two related classes, say `A` and `B`, and class `B` contains all the members of `A` and some additional ones. Is it reasonable to use `A` as a base class of `B`?

Of course, this situation indicates, that `B` *might be* really derived from `A`. But this is indication only that cannot replace the IS A – HAS A test. In 4.4, we have seen an example that leads to object composition. Here we give another example that will be solved by inheritance.

Example. Consider the particles in our Monte Carlo simulation. The interaction of electrically charged particles in the detector substantially differs form the interaction of uncharged particles, so it would be convenient to split the class of all the particles into two subclasses, one for the uncharged particles and the other for the charged ones.

The class representing the charged particles contains the same data members as the class representing the uncharged particles plus the `charge` attribute and the methods `setCharge()` and `getCharge()` to manipulate the charge. This might lead to the idea to define the `Uncharged` class representing the uncharged particles and use it as a base for the `Charged` class representing the charged particles. These two classes will serve as base classes for the classes representing concrete particle types (Fig. 6(a)).

This class hierarchy design is incorrect and leads to problems in the program. Suppose the following two declarations:

```
list<Uncharged*> ListOfUncharged;
Electron e;    //  Electron is charged particle
```

The `ListOfUncharged` variable is a double-linked list of pointers to the `Uncharged` instances. If the `Charged` class were derived from the `Uncharged` class, it would be possible to insert any charged particle into this container of uncharged ones. The following statement would compile and run (of course, the results would be unpredictable):

```
ListOfUncharged.push_back(&e); // It compiles...
```

The reason of this problem is evident — *the charged particle is not a special case of the uncharged particle* (the IS A test), so this inheritance is not applicable.

"Logical" Inheritance

Here we will show that the IS A – HAS A test may be insufficient in some cases. First, consider the following example.

Example. We will continue with the analysis of the Monte Carlo simulation of the charged and uncharged particles. The uncharged particles may be considered as a special case of the charged particles with the electric charge set to zero. Consequently, it seems to be logical to derive the `Uncharged` class from the `Charged` class (Fig 6(b)).

However, no member of the base class may be excluded from the derived class in the inheritance process. So the derived class, `Uncharged`, will contain the `charge` attribute and both the access methods. In order to ensure that the charge is zero, we have to override the `setCharge()` method so that it always sets the `charge` value to zero,

```
void Uncharged::setCharge(double ch) {
  charge = 0.0;         // Parameter value not used
}
```

Nevertheless, this construction may fail. Consider the following function:

```
void process(Charged& cp){
    const double chargeValue = 1e-23;
    cp.setCharge(chargeValue);
    assert(cp.getCharge() == chargeValue);
    // And some other code...
}
```

This is correct behavior of the `process()` function: It expects a charged particle, changes its charge to some predefined value and tests whether or not this change succeeded. If the argument is really a charged particle, it works.

However, the classes representing the uncharged particles, e.g., `Photon`, are derived from the `Uncharged` class and this class is derived from the `Charged` class, so the following code fragment compiles, but fails during the execution:

```
Photon p;        // Uncharged particle
process(p);      // Assertion fails...
```

This example shows, that even if the IS A test succeeds, it does not mean that the inheritance is the right choice. In this case, the overridden `setCharge()` method violates the contract of the base class method — it does not change the `charge` value.

6.2 Substitution Principle Revised

The preceding example demonstrates that under some circumstances the `Uncharged` class has significantly different behavior than the base class, and this leads to problems — even to run time errors.

This is the rule: Given the pointer or reference to the base class, if it is possible to distinguish, whether it points to an instance of the base class or of the derived class, the base class cannot be substituted by the derived class.

The conclusion is, that the substitution principle is more than a syntactic rule. This is a constraint imposed on derived classes, that requires, that the derived class instances must be programmatically indistinguishable from the base class instances, otherwise the derived class does not represent a subtype of the base class.

This conclusion has been originally formulated by Liskov (Liskov, 1988; Martin, 1996) as follows:

> What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is subtype of T.

Example. Let's finish the charged and uncharged particles problem. We have seen that the `Charged` and `Uncharged` classes may not be derived one from the other. To avoid both kinds of problems, it is necessary to split the hierarchy and to derive both classes directly from the `Particle` class:

```
// Proper Particle hierarchy
class Charged: public Particle { /* ... */ };
class Uncharged: public Particle { /* ... */ };
```

This class hierarchy is shown in the Fig. 6(c).

6.3 Inheritance and Encapsulation

In this subsection we demonstrate that the inheritance may lead to significant violation of the encapsulation, which may cause problems in the implementation of derived classes. We start with an example.

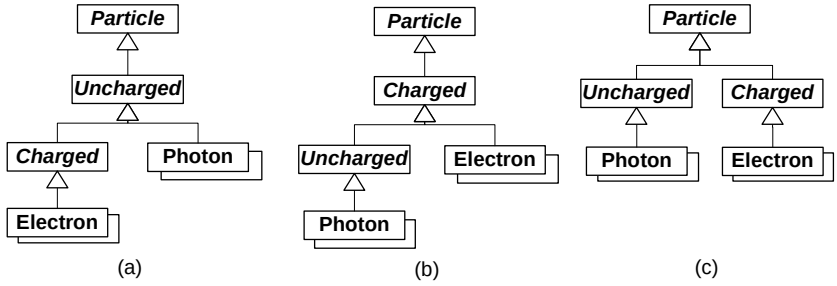

Fig. 6. Class hierarchies discussed in subsections 6.1 and 6.2. Only (c) is correct.

Example. The particles interact in the detector in different ways. Some of the interaction represent events that are subject to our investigation and need to be logged in the result file and further processed. (This means to record the particle type, energy, coordinates of the interaction etc.) But the events may appear in groups, so the `ResultFile` class will contain the methods `LogEvent()` and `LogEventGroup()`. The latter will get the vector containing data of several events as an argument. Suppose that both these methods are polymorphic.

At some later stage of the program development, we find that it is necessary to be aware of the total count of recorded events. The actual implementation of the `ResultFile` class does not support this feature and we cannot change it, e.g., because it is part of some program library.

The solution seems to be easy — we derive a new class, `CountedResultFile`, based on the `ResultFile`. The implementation could be as follows:

```
class CountedResultFile: public ResultFile
{
public:
    virtual void LogEvent(Event *e)
    {
        ResultFile::LogEvent(e);
        count++;
    }
    virtual void LogEventGroup(vector<Event*> eg)
    {
        ResultFile::LogEventGroup(eg);
        count += eg.size();
    }
private:
    int count;
};
```

The overridden methods simply call the base class methods to log the events and then increase the `count` of the recorded events.

It may happen that we find that the `LogEventGroup()` method increases the count of recorded events incorrectly: After the call

```
LogFile *clf = new CountedLogFile;
clf -> LogEventGroup(eg);   // (*)
```

the `count` value increases by twice the number of the events in `eg`.

The reason might be that the implementation of the `LogEventGroup()` method internally calls the `LogEvent()` method in a loop. This is what happens:

1. The (`*`) statement calls the `LogEventGroup()` method. This is a polymorphic method, so the `CountedResultFile::LogEventGroup()` method is called.
2. This method calls the base class `LogEventGroup()` method.
3. The base class method calls the `LogEvent()` method in a loop. *But because these methods are polymorphic, the method of the actual type, i.e., the* `ComputedResultFile::LogEvent()` *method is called.*
4. This method calls the base class method to record the event and increases the count of events. After that it returns to the `CountedResultFile::LogEventGroup()` method. This method increases the event count once again.

To implement the derived class properly, we need to know that the `ResultFile::LogEventGroup()` method internally calls the `ResultFile::LogEvent()` method. *But this is an implementation detail, not the part of the contract of the methods of the* `ResultFile` class.

Solution

This problem may easily be avoided by using interfaces and object composition (cf. class diagram in Fig. 7); it is necessary to use another design of the `ResultFile` class, as well as another design of the `CountedResultFile` class.

First we design the `ResultFileInterface` interface as follows:

```
class ResultFileInterface {
public:
     virtual void LogEvent(Event *e) = 0;
     virtual void LogEventGroup(vector<Event*> eg) = 0;
};
```

The class `ResultFile` will implement this interface:

```
class ResultFile: public ResultFileInterface {
 // Implementation as before
};
```

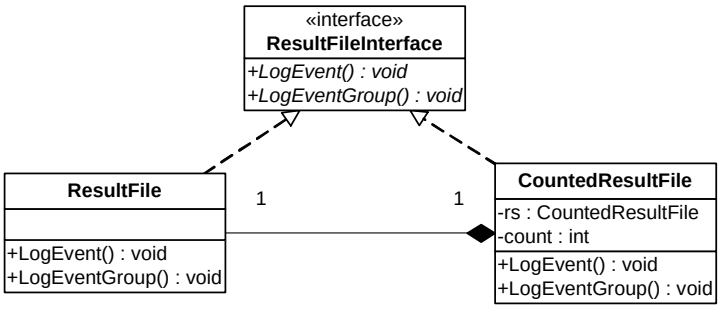

Fig. 7. Class diagram of the correct design. Only interface methods and corresponding attributes are shown.

Now, `CountedResultFile` may be designed as an independent class that implements the `ResultFileInterface` and uses the `ResultSet` as an attribute:

```
class CountedResultFile: public ResultFileInterface {
public:
    virtual void LogEvent(Event *e)
    {
        rs.LogEvent(e);
        count++;
    }
    virtual void LogEventGroup(vector<Event*> eg)
    {
        rs.LogEventGroup(eg);
        count += eg.size();
    }
private:
    int count;
    ResultSet rs;
};
```

The problem may not appear, because the `CountedResultFile` is not derived from the `ResultFile` now. Nevertheless, they may be treated polymorphically, i.e. instances of the `CountedResultFile` may be used instead of instances of the `ResultFile`, if they are used as instances of the `ResultFileInterface` interface.

7 Structure of the Object Oriented Program

We conclude this chapter by describing shortly the typical structure of the OOP program.

As we have seen in previous sections, an OOP program consists of objects that collaborate by messages. In this structure, one object must play the role of a *starting object*. This means that one of the methods of this object will be called as the program start. The starting object typically creates other objects in the program and manages the their lifetime.

All the other objects represent various parts of the problem solved and are responsible for the associated resource management, computation, etc.

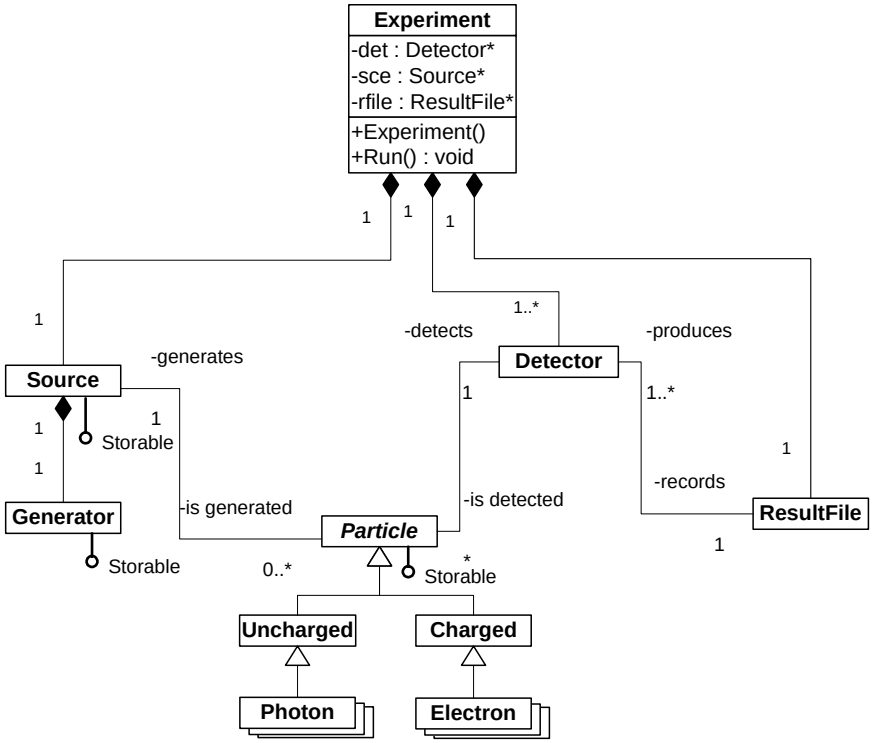

Fig. 8. Basic structure of the simulation program.

Example. Here we finish the Monte Carlo simulation of an experiment with particles. We have already mentioned the `Experiment` class covering the application as a whole. The only instance of this class in the program will be the starting object. The `Experiment` class will contain the public method `run()` that will represent the run of the experiment.

As we are in C++, our program must have the `main()` function where the program starts. It will create the starting object and let it run:

```
int main() {            // Create the starting object
```

```
    Experiment().Run(); // and run it
}
```

The `Experiment` class could be defined as follows:

```
class Experiment{
public:
    Experiment();       // Constructor
    void Run();         // Run the experiment
private:
    Detector *det;      // Detector in this experiment
    Source *sce;        // Particle source
    ResultFile *rfile;  // Result file
 };
```

The `Experiment::Experiment()` constructor reads the input data (e.g., cross sections describing the probabilities of the interactions) and creates and initializes the attributes (particle source, result file etc.).

The `Experiment::run()` methods does the work — it starts particle emission in the source using the appropriate method of the `Source` class, determines, whether the given particle hits the detector using the appropriate `Detector` method, records the results of the detection into the `ResultFile` instance and in the end processes the results using the appropriate `ResultFile` methods.

Class diagram of the program at the level we have given here is presented in Fig. 8.

Note that this is top-level design only. The extent of this chapter allows us to demonstrate only the beginning of the object oriented approach to the example.

8 Conclusion

Attempts to formalize the process of object oriented analysis and design have been made since the beginning of the OOP. A widely used approach is described in Booch (1993).

In object oriented design some common problems — or tasks — may appear in many different situations. Reusable solutions of these problems are called *Design Patterns*. A well known example of design pattern is Singleton — the class that may have at most one instance. Another example is Responsibility Chain; this design pattern solves the problem how to find the proper processing of varying data, even if the way of processing may dynamically change.

The idea of design patterns and the 23 most common design patterns in OOP are described in Gamma (1999).

References

Booch, G.: *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, New York (1993).

Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide.* Addison-Wesley, New York (1999).

Gamma, E., Helm, R., Johnson R. and Vlissides, J: *Design Patterns.* Addison-Wesley, New York (1995).

Liskov, B.: *Data Abstraction and Hierarchy.* SIGPLAN Notices, **23**, 5 (May 1988).

Martin, R. C.: *The Liskov Substitution Principle.* The C++ Report, March 1996.

Meyer, B.: *Object-Oriented Software Construction.* Prentice Hall, New York 1997; see also
http://en.wikipedia.org/wiki/Eiffel_programming_language

Stroustrup, B.: *The C++ Programming Language. Third edition.* Addison-Wesley, New York (1998).

Index

access specifier, 7
attribute, 2

base class, 9

class, 3
class diagram, 7
class hierarchy, 10
class interface, 7
constructor, 6

destructor, 6

early binding, 16
encapsulation, 2

getter, 2

inheritance, 9, 14
instance, 3
interface, 3, 15, 19
interface for derivation, 7

late binding, 16

message, 3
metaclass, 4
method, 2
model domain, 1
Monte Carlo method, 2

object, 2
object composition, 6, 14
Object oriented programming, 1
object, starting, 28
OOP, 1

polymorphic class, 16
polymorphism, 15
problem domain, 1

random number generator, 6
random numbers, 6

setter, 2

UML, 2, 7