

Chapter 1

Introduction

1.1 Aim of Object-Oriented Design and Programming

The aim of any change in design and programming tools is usually driven by requirements of increased efficiency and power at lower cost. To illustrate the need for increased efficiency, consider the 1979 US Government Accounting Office report on software expenditure summarized in table 1.1

Percentage of money spent on software products	
paid for but not delivered	47%
delivered but never used	29%
abandoned or reworked	19%
used after change	3%
used as delivered	2%

Table 1.1: 1979 US Government Accounting Office Report (FGMSD-80-4) on software expenditure for 9 federal software projects.

Since then, the emphasis on big software projects has shifted towards a modular approach similar to that found in hardware systems. Teams of software engineers and programmers work on large software projects by splitting the work into separate modules with predefined interfaces and optimal independence. For example, the parts in a PC are replaceable. You can replace your old monochrome monitor and display driver card with a super-VGA card and monitor without any affect on the rest of the system. The VGA card interfaces with the rest of the system via a standard bus and a design change in, for example, the processor, does not necessarily effect the manufacturers of display or hard disk systems. Furthermore, if the system is upgraded (e.g. the motherboard and processor) one can reuse most of the old components (e.g. screen, keyboard, power supply, disk drives, etc.)

Object-oriented design and programming methods aim to achieve the following:

- To simplify the design and implementation of complex programs.
- To make it easier for teams of designers and programmers to work on a single software project.
- To enable a high degree of reusability of designs and of software codes.
- To decrease the cost of software maintenance.

In order to achieve these aims, object-oriented programming languages are expected to support a number of features:

1. Information hiding (encapsulation)
2. Polymorphism
3. Inheritance

Each of these concepts is discussed below.

1.1.1 Information Hiding

Information hiding is achieved by restricting the access of the user to the underlying data structures to the predefined methods for that class. This shifts the responsibility of handling the data fields correctly from the user of the code to the supplier. For example, a programmer using a `Date` class is usually not interested in the implementation details. These include the underlying data structure of `Date` (i.e. whether the month is stored as an integer, a string or an enumeration type) and the underlying code (i.e. how two dates are subtracted from each other). If, at a later stage a more efficient way of storing the date or of calculating the number of days between two dates is introduced, this should not affect the programmers who have been using the date class. They should not have to search through all their programs for any occurrence of `Date` and to make the relevant changes. Such a maintenance nightmare and the very high cost accompanying it is prevented by information hiding where access to the underlying data structures is given only via the predefined methods of the class. A further advantage of information hiding is that it can be used to guarantee the integrity of the data (e.g. to prevent the user from setting the month equal to 13).

1.1.2 Inheritance

Often one object shares a number of characteristics with another, but has a few additional attributes. For example, your firm might have a data base for vehicles. For each vehicle the registration number, model, year and maintenance history are stored. Suppose the firm wants to expand this data base to include its trucks. However, they want to store additional information for trucks, such as the the pay-load and the number of axes (for toll-road purposes). Instead of rewriting all the code for trucks, one can see a truck as a special case of a vehicle (having all the attributes of a vehicle plus a few additional attributes). By making truck a derived class of vehicle, it inherits all the attributes of vehicle and the programmer has only to add the code for the additional attributes. This not only reduces the amount of coding to be done and maintained, but it also ensures a higher level of consistency. Should the country decide to change its format for number plates, it has to be only changed in the base class vehicle and the trucks automatically inherit the change.

1.1.3 Polymorphism

Often one wants to perform an action such as editing on a number of different objects (i.e. text files, pixel files, charts, etc.). Polymorphism allows one to specify an abstract operation like editing, leaving the actual way in which this operation is performed to a later stage. In the case where dynamic binding (linking during run-time) is used the decision of which code to be used for the editing operation is only made during run-time. This is especially useful in the case where the actual type of object which is to be edited is only known at run-time.

Furthermore, if a new object type is to be supported (e.g. faxes) then the new editing code (provided by the fax-software supplier) can be linked in at run-time. Even when the original program is not recompiled, it can support future additions to the system. Polymorphism is hence again a higher level of abstraction, allowing the programmer to specify an abstract action to be performed on abstract objects.

1.1.4 Templates

A less abstract form of polymorphism is provided via template support. In many instances a similar algorithm is needed for many different data structures. For example, sorting algorithms are essentially the same for integers, floating point numbers, names or any other data structure. Similarly, matrix multiplication is essentially the same for integers, floating point numbers and complex numbers. Traditionally one would have to write a sorting or matrix-multiplication algorithm for every data type. Any change in the algorithm must then be implemented in all the copies of that algorithm. Templates allow one to define a single algorithm for many different data types. The actual data types are resolved during compiling and hence there are no run-time overheads.

1.2 Why C++ ?

Firstly, C++ supports the central concepts of object orientated programming: encapsulation, inheritance and polymorphism (including dynamic binding). It has good support for dynamic memory management and supports both, procedural and object-orientated programming.

However, other well designed programming languages have failed against relatively poor competitors. Being a good programming language is not sufficient for survival. An additional important requirement for a powerful programming language is portability. If a firm replaces one computer system with another, only a minimal amount of recoding (if any) should be required. C++ compilers are available for virtually all machines and a high level of compatibility is ensured by the ANSI-standard for C++. Furthermore, there exists a huge amount of good C-code. C++ supports C as a subset and hence this huge amount of code can be reused in new software projects.

All these points have contributed to C++ being the fastest growing computer language for nearly all computer and operating systems and for nearly all software applications, ranging from scientific to administrative programs to real-time industrial applications and computer games.