Object-oriented Programming in Crystallography

D. S. Moss
Crystallography Department, Birkbeck College, Malet Street, London WC1E 7HX, UK
d.moss@mail.cryst.bbk.ac.uk
http://www.cryst.bbk.ac.uk/~ubcg05m

W. R. Pitt
Crystallography Department, Birkbeck College, Malet Street, London WC1E 7HX, UK
w.pitt@mail.cryst.bbk.ac.uk
http://www.cryst.bbk.ac.uk/~ubcg08l

Abstract

An object-oriented class library, designed for use in bioinformatics and molecular modeling is being developed at Birkbeck College. Although the library is not targeted directly at crystallographers, the methods provided are useful to anyone analyzing molecular structures and sequences. The library can be used to convert a Protein Data Bank (PDB) format file into Protein and Atom objects. Once converted, methods of the Protein and Atom classes can be applied to the data. For instance, one can calculate the internal geometry of a protein structure and perform transformations on its atomic coordinates. The library is in the early stages of development but will serve to introduce crystallographers to the object-oriented programming paradigm and how it can be applied to biocomputing.

1 Introduction

In the late 1950s crystallographers were among the first scientists to make use of computers. In fact much of the crystallographic computing software that we use today has its origins in the 1960s when 32k was a typical core memory size of computers. Since then there have been increases of several orders of magnitude in both the memory size and speed of computers. Crystallographic software has taken advantage of these developments which have, for example, permitted electron density maps to be stored in central memory

Developments in software engineering have also taken place, particularly in the past ten years. Arguably the most significant development has been the object-oriented approach to software construction and its support in new programming languages such as C++ and Java. However, crystallographic computing with its large base of legacy code, is only just beginning to exploit these new developments.

Many other changes in software development have taken place over the last decade. In the 1980's, scientific programming was done almost exclusively in Fortran, file formats were relatively simple, graphical user interfaces had not emerged as an important issue and the explosive growth of macromolecular crystal structures had only just started. Today C has become one of the most widely used languages for scientific computing and the Internet and the World Wide Web have transformed the way in which the scientific community can participate in software development and gain access to the results.

Object technology has recently gained widespread acceptance in software development, not just because of code reusability, but because it takes advantage of a new generation of object-oriented environments. These environments and associated tools are rapidly becoming standardized. A draft ISO/ANSI C++ standard[1] has been published and the Standard Template Library (STL)[2], which forms part of it, is likely to influence programming paradigms for years to come. The Common Object Request Broker Architecture (CORBA)[3], which allows the distribution of objects across disparate systems, is becoming an industry standard and the Object Database Management Group (ODMG)[4] has set a standard for object databases. On the World Wide Web, the Java[5] language which is almost exclusively object-oriented, is already having an important impact on the engineering of Web software.

1.1 Previous Work

An early application of Object-oriented programming (OOP) to biomolecular computing was published in 1990[6]. Gray *et al*. created an object-oriented database for protein structure analysis. To this day, protein structure data is mainly stored either in unstructured files or in relational databases. Gray *et al*. discuss the short fallings of these more conventional methods of storing data when applied to sequentially structured data which is likely to be subject to complex and unpredictable queries. Their database overcomes these short fallings but has not been made widely available and can only be queried using the little known programming languages PROLOG and Daplex.

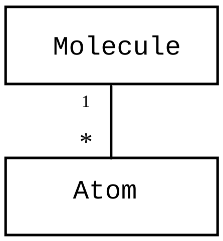
A more recent application of OOP to the analysis of protein structure data[7] uses the more widely adopted language C++. The product of this work is a macromolecular class library called PDBlib. This library is similar to the one being developed at Birkbeck College. The two libraries differ in that the Birkbeck version is designed for sequence as well as three dimensional molecular structure analysis. Significantly, PDBlib predates the release of the draft ISO/ANSI standard for C++.

2 Object-Oriented Programming Explained in Brief

OO programs manipulate abstract representations of the entities that are being modeled. These representations are the *objects*. A *class* defines one type of object. Objects contain data (*member data*) and the methods (*member functions*) that can be used to manipulate this data. This binding together of data and methods within an object is called *encapsulation*.

One possible class is a unit cell which could have cell dimensions, space-group and molecules as data members. Member functions of the unit cell class could be written to calculate the volume and density or to carry more complicated procedures such as an energy minimisation of its contents.

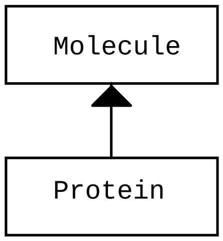
Objects can contain, or have, other objects. For instance, a unit cell object can contain molecule objects which in turn can contain atom objects, and so on. The relationship between the unit cell class and the molecule class and between molecule class and the atom class is called an *association*. This is hierarchical structure makes its easy to create intuitive abstract models of biological molecules.



Association: A Molecule can *have* one or more Atoms[8]

Possibly the most attractive feature of OOP to software library developers is that is specially designed to ease the generation of reusable code. It does this by providing the means whereby classes can be written that are intended to be used as given and left unchanged by users of a library. Developers of applications should only change such classes by extend their behavior. This restriction means that classes can be created that behave in a reliable manner.

Users who wish to add their own data and functions to a class should do so by creating a new class and which *inherits* from the original. If, for example, a molecule class existed in a library and a user wanted to add features to this class that are specific to protein molecules, then they could create a new class called protein which inherits from the molecule class. In this way the new class will have all the features of the molecule class plus what ever protein specific functions and data that user wants to add. In this case the molecule class is the *base class* and protein class is the *subclass*. A protein object *is a* molecule object but the reverse is not true.



Inheritance: A Protein *is a* Molecule[8]

A user of a well designed class library only has access to the member functions of an object and can only access the data members via these functions. This restriction prevents users from carrying out inappropriate operations on the data and thus makes the library more stable. It also means that the authors of the library are free to change the underlying data structures and algorithms without users of the library having to change their programs.

3 A Class Library for Biomolecular Computing

It is our aim to provide software developers in the field of biomolecular computing with a well tested, efficient and useful library of reusable software. This software will carry out certain basic operations such as reading in files of various common formats and calculating molecular geometry, freeing a the developer to concentrate on less mundane tasks. The library will also include tried and tested algorithms such as PROCHECK[9]. Thus, the library will facilitate the combination of hitherto disparate applications and allow the user customize these applications.

3.1 The Choice of Programming Language

The aims described above are best fulfilled by the use of OOP as this programming paradigm is designed for the production of reusable and extensible code. It also enables the production of code that is more intuitive to scientists who are not experienced programmers.

There are a number of OOP languages available, the most commonly used ones are C++, Java, Smalltalk, Delphi and Eiffel. Two of these, C++ and Java, are by far the most commonly used in biocomputing.

Because of its provenance, C++ has significant non object-oriented content. Java, however, is an almost pure OO language. With this language, small applications (applets) can be written that can be downloaded via the Internet and executed within an Internet browser. This provides the means whereby applications written in Java can be made extremely accessible and user friendly. However, no international standard exist for Java and execution speed was not a high priority in its design. Standards are important if packages written in a language are to be portable across many platforms. Execution speed is important in biocomputing where long, central processing unit (CPU) intensive computing jobs are common.

We decided that the best language to use is C++ as it is still probably the most widely used OOP language amongst scientists. C++ is an extension of C, which is a language a large number of scientific programmers currently use. It is easy to learn C++, given a knowledge of C. Also, one can write extremely efficient code in C/C++. Perhaps the most important reason for choosing C++ is that a ISO/ANSI draft standard for this language was released in April last year. This should enable us to produce code that can be understood by any C++ compiler, so it can be used on practically any hardware platform.

We do, however, intend to make use of Java in writing user interfaces to our library and to applications written using our library. We also intend to publish the design of our library in the form of language independent class diagrams. This will make it easy for a version of our library to be written in Java or some other language, should the need arise.

3.2 Other Design Issues

Before we started writing any code we created an initial design for the library using class diagrams. Once we started implementing this design we discovered undesirable features in it. These features were removed and the design altered. In this way the library evolves through a cycles of design and implementation.

Many design decisions are compromises but here is a list of our priorities:

1. Ease of use
2. Efficiency of execution
3. Extensibility
4. Modularity
5. Functionality

To make the library easy to use the classes must define types that are intuitive to users from the biological and chemical sciences. These classes should also have names that easily understood. For example, we have classes called Protein, Atom, TorsionAngle, CovalentBond, Residue, and AminoAcidResidue.

Although computers are getting faster and faster, researchers will always push their machines to their limits. This means that these people demand programs that run as efficiently as possible. Certain features of C++ can slow programs down. For example, methods in classes that are at the bottom of a many layered inheritance tree often do not execute as quickly as those in classes at the top of a tree. This has an impact on the design of the library.

For the library to be of widespread use, it must be designed in such a way that application developers can extend the behavior of classes. Also, these developers do not want classes that are weighed down by functions and data that they do not want to use. This means that we must carefully consider whether each data member and member function is absolutely necessary before adding it to a class. This is especially true for classes high up in an inheritance tree.

The inclusion of useful functions that are time consuming or not easily written within a library will obviously make it more attractive to application developers. However, what would seem useful to someone writing sequence analysis program may not be so appealing to an author of molecular mechanics package. For this reason there should not be unnecessary interdependencies between classes within the library.

3.3 The Current Stage of Library Development

Although the library is, at time of writing, in early development it has reached the stage when it can be of some use. It can be used to read in a Protein Data Bank (PDB) format, extract the residue, atom, and bond information from it and create the appropriate objects.

Once these objects are created, the data read from the file can be inserted into them. After file processing is completed, the user can manipulate the objects using their member functions. The user can then send information about objects to the screen or write out a new PDB file at any time.

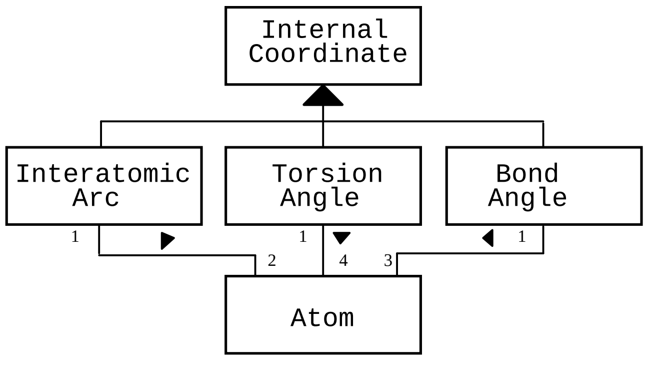
Currently, relatively few member functions exist. The user can however perform transformations on the coordinates of atoms, calculate internal coordinate geometry, and calculate the root mean squared deviation (RMSD) between two molecules. These simple calculations would require considerable effort to write a program from scratch to carry them out.

At the moment work is concentrated on building the frame work of the library rather than adding new functionality. We have decided that library should be split into several largely independent sub-libraries, these being:

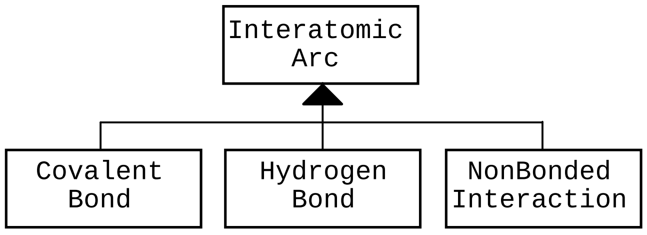
1. A library for manipulating molecules at the atomic level.
2. A library for manipulating molecules at the sequence level
3. A specialized mathematics library
4. A library of specialized data structures

Many applications and classes in the library will use more than one of these sub-libraries, but this modularity should reduce the inclusion of superfluous functionality to a minimum.

Most of our library will be made up of the first two sub-libraries. The following class diagram illustrates the part of the library that models the internal coordinates of a molecule.



InternalCoordinate Classes and their association with the Atom class[8]



InteratomicArc Classes[8]

The diagrams above show that the classes InteratomicArc, TorsionAngle, and BondAngle are subclasses of the InternalCoordinate class. An InteratomicArc links 2 Atoms, A TorsionAngle links 4 Atoms, and a BondAngle links 3 Atoms. The class InteratomicArc is the base class for CovalentBond, HydrogenBond and NonBondedInteraction classes.

Our math sub-library contains classes for manipulation of matrices and vectors in ways that are common in biocomputing. The basic mathematics operations such as function for calculating square roots and cosines already exist in the standard C++ math library.

Some very efficient, flexible, reliable and easy to use classes for storing and manipulating data already exist in the form of the Standard Template Library (STL). This class library is part of standard C++ and can be used to store any sort of data, including objects, in lists and other containers. The STL also includes generic functions for searching, sorting and other operations on these containers. We employ the STL heavily and have very little need to create our own data structures.

3.4 Future Developments

There is much work to be done to build, test and document the library. We also intend to write some applications to illustrate how very powerful programs can be written using the library with relatively few lines of code.

We hope that the development of the library will become more of a collaborative effort. The use of existing programs written in C and Fortran means that the list of contributors to the library is growing rapidly. However, when the first version of the library is released, we expect that researchers will take classes in the library and extend them, through inheritance, so that they fulfill their specific needs. It is hoped that these researchers will allow us to put their new classes into the library. Thus, if all goes to plan, the library will branch out to cover more and more specialized areas of biocomputing.

We intend to release the first version of the library in February 1997. When the current funding of the project runs out at the end of 1998, training in and maintenance of the library will be taken over by the CCP11 consortium. This group is based at Daresbury and exists to foster the role of bioinformatics within the British academic community[10].

4 Summary and Conclusions

We feel that, given its widespread use in the software industry sector, OOP will become increasingly popular amongst scientists. This paper serves to introduce crystallographers to the concepts of OOP as applied to biomolecular computing. To help software developers in this field to develop OO code we are writing a class library designed specifically for their use. This library will aid the development of more sophisticated, yet stable programs by providing well tested and efficient classes which can be reused and built upon.

References

[1] see http://www-leland.stanford.edu/~iburrell/cpp/std.html

[2] see http://weber.u.washington.edu/~bytewave/STL.html

[3] see http://www.omg.org/corba.htm

[4] see http://www.odmg.org/

[5] see http://java.sun.com/

[6] P. M. D. Gray, N. W. Paton, G. J. L. Kemp, and J. E. Fothergill, "An object-oriented database for protein structure analysis" Protein Engineering, Vol. 3, No. 4, pp. 235-243, 1990.

[7] W. Chang, I. N. Shindyalov, C. Pu, and P. E. Bourne, "Design and application of PDBlib, a C++ macromolecular class library" CABIOS, Vol. 10, No. 6, pp. 575-586, 1994.

[8] Class diagram in the Unified Notation: G. Booch and J. Rumbaugh, "The Unified Method for Object-Oriented Development", see http://www.rational.com/ot/uml.html

[9] R. A. Laskowski, M. W. Macarthur, D. S. Moss and J. M. J. Thorton "PROCHECK: a Program to Check the Stereochemical Quality of Protein Structures" Appl. Cryst. Vol. 26, pp. 283-291, 1993.

[10] see http://gserv.dl.ac.uk/CCP/CCP11/main.html