THE OBJECT TECHNOLOGY FRAMEWORK (OTF): AN OBJECT-ORIENTED INTERFACE TO MOLECULAR DATA AND ITS APPLICATION TO COLLAGEN

GREGORY S. COUCH, CONRAD C. HUANG, ERIC F. PETTERSEN,
THOMAS E. FERRIN[1]
Computer Graphics Laboratory
University of California, San Francisco
San Francisco, California 94143-0446 USA

ALLISON E. HOWARD and TERI E. KLEIN[*]
Department of Pharmaceutical Chemistry
University of California, San Francisco
San Francisco, California 94143-0446 USA

This paper describes the OTF software system developed at the University of California, San Francisco Computer Graphics Laboratory for creating powerful C++ classes that facilitate rapid biomolecular application development and its application to collagen modeling. C++ class libraries for accessing and manipulating data from standard scientific data sources can be generated from the program *genlib* and its class library toolkit *Molecule*, thereby making this functionality much more accessible to application tool developers. The application of the OTF in the development of the program **gencollagen**, which generates model collagen structures, is described. The source code for the OTF is freely available at http://www.cgl.ucsf.edu/otf/ to interested application developers.

1 Introduction

1.1 General Overview: The Problem

A researcher developing a molecular analysis software tool often needs to provide many functions besides the analysis itself, including access to standard databases, representation of external data in an internally usable form, basic bookkeepping functionality such as atom deletion or bond list traversal, and convenient interchange of data with other tools. These functions, while necessary and desirable, are really not of main interest to the researcher, who would generally prefer to simply do the analysis of the data than write these associated support functions. This frequently results in the creation of molecular analysis tools with minimal support functions that utilize their own custom data formats or some subset of a standard format. They frequently cannot handle data that uses the full range of a standard format and cannot easily interchange data with other tools. In addition, some time is typically spend coding and debugging what support functions do exist, further slowing down tool development. The end result for the tool *user* is an environment where tool cooperation is very difficult to achieve, and where fewer tools exist due to the difficulty of developing them.

[1] Corresponding author for OTF

1.2 A Solution: The Object Technology Framework (OTF)

The OTF software system was developed to address these problems. The library classes generated by the OTF provide the functionality needed to access standard data sources and to carry out operations typically performed on molecular data. The classes accelerate application development and enhance interoperability.

The OTF organizes an application's molecular data into natural abstractions such as atoms, residues, and molecules, encapsulated as C++ classes. The application developer specifies what OTF-supported functionality is required and the OTF automatically generates classes that incorporate the designated features. No unwanted functionality is included.

The application developer will typically also want to extend the classes to support application-specific data and functionality. Such extensions are specified in the same format that OTF features are specified (C++ code segments embedded in identifying tagged delimiters). Consequently, if the programmer creates extension functionality that is of general utility, it is easy to contribute the functionality back into the OTF as a new feature for use with future applications.

There are standard C++ mechanisms for providing mix-in functionality, customizability, and re-usability as described above. These mechanisms for the most part work well in the context of single application or closely-related suite of applications~~,~~. For example, Democritos[13], PDBlib[14], and BTL[15] are biomolecular C++ class libraries that offer a broad spectrum of functionality. An application can use one of these libraries and customize and extend the library's classes for application needs. However, due to shortcomings with standard C++ extension mechanisms (discussed in Appendix 1), it is difficult or impossible to re-integrate any extended functionality back into the library for use with other applications. The OTF allows easy re-integration of useful functionality

The development of the OTF has been motivated by the realization that biomolecular application developers were constantly re-inventing functionality present in other applications and that if such functions could be offered as easily re-usable components that could be simply integrated into custom applications, much time and effort would be saved. The functionality would be tested, complete, and consistent from application to application. The shortcomings in C++ library re-usability stood in the way of these desirable goals, and hence, the OTF was developed to offer a way to make functionality generally re-usable.

1.3 Applications

The OTF has been used to develop several applications both locally and abroad. Locally, applications developed include: (1) *autoindex*, a tool for generating an HTML index of a directory; (2) *gneighbor*, a tool for computing and printing the Gaussian neighborhood values for protein residues; (3) *phipsi*, a tool for computing and printing protein ϕ, ψ, ω, χ angles and (4) *gencollagen*, a tool for generating model segments of collagen. In this paper, we will describe the details of the *gencollagen* application.

2. Design Concepts

The major classes that the OTF currently offers built-in support for are:

Atom, *Bond*, *Residue*, and *Molecule*. The functionality that the OTF can provide with the above classes is separated into coherent groups referred to as *toolkits*. The functionality each toolkit offers is further divided into one or more components that can be used on an individual basis. Application developers using the OTF run a program called *genlib* to compose exactly those components they want in their application into the above classes. An important difference between OTF components and the standard C++ inheritance mechanism is that components may cross class boundaries. For example, the *CoordSet* component, used for managing multiple atomic coordinate sets, augments the interface of both the *Molecule* and *Atom* classes.

Each component has an interface and one or more implementations. The interface specifies what functionality the component offers to the application (classes, member functions, *etc.*) and each implementation contains all the code necessary to completely implement the interface. The reason that some components offer multiple implementations is so that if there is an important design trade-off in the implementation (*e.g.*, speed *vs.* memory use) then implementations embodying differing selections in the trade-off can be provided. Application developers select a particular implementation when selecting a component for inclusion in their application. The separation between interface and implementation enables the developer to create an application quickly using existing code and to optimize performance later by replacing bottleneck implementations.

In order to customize the generated classes for the application's specific needs, the developer writes C++ code that will be composed by *genlib* into the class at the same time as the selected components. The C++ code is written in the same fashion that the standard OTF components are written, which is regular C++ code embedded in simple delimiters that *genlib* can recognize. Consequently, if the customizations are of some general utility, they can be organized into components and toolkits which can be easily re-used in future applications.

The classes generated by *genlib* are designed to be used *as is* and should not be used as base classes. This is because the generated classes have containers referring to other generated classes (*e.g.*, the *Molecule* class has containers of both *Atom*s and *Bond*s). In order for the containers to refer to subclasses, the base classes would need to be template classes. Template base classes are undesirable for reasons discussed in Appendix 1. The *genlib* mechanism is provided so that the classes' functionality can be extended while avoiding subclassing.

2.2 *Molecule* Class Library Toolkit

The *Molecule* class library toolkit offers abstractions of molecular data as atoms, residues, and molecules via C++ classes. These classes can be customized and extended by the application developer using *genlib* described in *section 2.3*. The molecular properties handled automatically by these base classes are:

Atom
- name
- atomic number
- bonding environment, *e.g.*, planar vs. tetrahedral
- index into coordinate set

Residue
- type
- sequence number

Molecule
- coordinate sets

- comments

Atomic coordinates are not directly associated with atoms, but instead molecules can have zero or more *coordinate sets*, which enumerate alternative conformations for the atoms. A molecule may have different conformations when bound verses unbound, or, multiple conformations over the course of a computational simulation. Multiple coordinate sets are essential for molecular dynamics applications and magnetic resonance spectroscopy analysis. The concept of a "current" coordinate set is maintained, and it is simple to retrieve an atom's coordinates from the current set.

The *Molecule* class library toolkit includes a variety of miscellaneous support classes, including *MolComment* for annotating a molecule, and utility functions, such as *tmplAtomicNumber* for converting atomic symbols to atomic numbers. An important "miscellaneous" included functionality is the conversion of non-standard atom names to IUPAC-IUB standard nomenclature [3].

A very important set of support classes are those that provide for reading from and writing to standard data sources. Currently, a class for Protein Data Bank format [4-6] is provided.

2.3 Genlib

A utility called *genlib* is provided for assembling the *Molecule* toolkit components into custom C++ classes usable by an application. *Genlib* takes a small configuration file as input and assembles desired components into a compiled class library. The configuration file guides application-specific class customizations: instance variables can be added to handle additional molecular properties, new member functions can be introduced to provide needed functionality, and C++ constructor and destructor code can be added to handle necessary set up and disposal operations. Unneeded components can be omitted. In addition, the configuration file allows the developer to specify, on a case by case basis, how bulk data in a class should be organized. For example, the application designer can specify that residues in a molecule should be organized using a container class with array-like semantics, rather than the default container class that uses ordered set semantics.

Genlib's code assembly capabilities greatly speed application development by reducing coding time and eliminating typographical errors as well as conceptual errors related to class integration. *Genlib* also provides a consistent organization across the assembled classes.

3. Example Application: **gencollagen**

We developed the program **gencollagen** in order to model collagen for structural studies related to the disease *Osteogenesis imperfecta*, and to model novel collagen-like biomaterials. **Gencollagen** generates a Protein Data Bank (PDB) file [4-6] containing the idealized atomic coordinates for a triple-helical fibril collagen fragment of twelve amino acids or more, given the protein sequence and several additional input parameters. The program's amino acid internal coordinates were obtained using AMBER-94 [8] optimized geometries. The default values for the helical and symmetry parameters are from Miller *et al*. [9]. **Gencollagen** constructs molecules in the following manner: (1) read the amino acid data file, (2) read the input file, (3) create the backbone atoms for a single chain [9, 10], (4) compute the triple-helical axis based on the single chain [10], (5) create the backbone atoms for the other two chains from symmetry operations, (6) order the amino acids for sidechain addition (C-terminus $\rightarrow$ N-terminus, N-terminus $\rightarrow$ C-terminus or completely random), (7) add sidechain atoms using a rotamer library (Dunbrack and Karplus [11], Ponder and Richards [12] or random) and (8) write the output PDB file.

The coding of **gencollagen** was facilitated by the use of the OTF as illustrated by the following statistics:

(1) Total lines of user-written code:	2000
(2) Lines of code for high-level control logic:	300
(3) Lines of code related to computing the geometry:	600
(4) Lines of code for managing AMBER-94 geometries:	500
(5) Lines of code for managing **gencollagen** parameters:	500
(6) Lines of code for miscellaneous functionality	100
(7) Lines in gencollagen configuration file[1]:	31

Lines 4 and 5 cover application-specific data management (not assisted by the OTF). Of the remaining code, approximately 90% (lines 2 and 3) was written specifically for generating collagen models (*e.g.*, computing helical symmetry, adding sidechain atoms, duplicating chains). Only Line 7 and parts of Line 6 were devoted to managing standard molecular data; the OTF handled the bulk of such management (*e.g.*, data organization into molecules, residues, and atoms; creation of the output PDB file). Considering that 1000 lines of code were necessary for management of application-specific data, clearly much coding effort was saved by using the OTF for standard molecular data functionality.

For smaller analysis programs that require only molecular data, the benefits of the OTF are even more evident. For example, *phipsi* (program for computing protein bond angles) required 197 lines of C++ code and 33 lines of *genlib* configuration; *pdbiv* (program for converting a PDB file into an Open Inventor graphics file) required 177 lines of code and 2 lines of *genlib* configuration; and gneighbor (program for computing the Gaussian neighborhood of residues) required 172 lines of code and 30 lines of *genlib* configuration. The ability to quickly build molecular analysis tools is an important use of the OTF.

Acknowledgments

Development of the Object Technology Framework (OTF) was supported by the National Science Foundation's Knowledge and Database Systems Program (grant IRI-9116999) and by the National Institutes of Health National Center for Research Resources (grant 2P41-RR01081).

Appendix 1: Inadequacies of C++ for Generic Class Library Development

This appendix delves into the minutiae of why C++'s normal language facilities are inadequate for general-purpose class library development, which necessitated the development of *genlib*. This section will only be understandable to those already thoroughly familar with C++'s language facilities.

Trying to provide the OTF functionality in a set of simple base classes has two major problems. One problem is that the *Molecule* base class wants to contain instances of the class derived from the *Bond* base class, for example, but only knows about the base class. This means that *Molecule* member functions returning "Bonds" can only return the *Bond* base class instead of the derived class. This leads

[1]Shown in full in Appendix 2.

to tedious error-prone casts from the base class to the derived class throughout the application. The other problem is that these base classes cannot create instances of the derived classes. This is crucial when attempting to read in a PDB file, for example, where *Atom*s, *Bond*s, *etc*. need to be created.

Using template base classes would solve the problems just described. However, it suffers from three serious deficiencies that *genlib* alleviates:

- The `template` statement amplifies the broken encapsulation of the `friend` statement.
- True modularity cannot be achieved.
- The resulting class library is daunting in its complexity, both to the library designer/implementer and to the application-programmer end user.

Template/friend Interaction

The first problem is that if a template base class provided by a class library needs to declare friend classes or functions, then encapsulation breaks badly. A friend declaration normally breaks encapsulation to a limited extent in that the class given friendship can meddle with the private sections of the granting class. With standard friendship, the addition of new (but unrelated) functionality to the friend class requires no change to the friendship-granting class. But if the classes are template classes, this is no longer the case.

This is probably most easily understood with an example. Assume that MoleculeBase and AtomBase are the template base classes for the application's derived classes Molecule and Atom, respectively. Given that class MoleculeBase needs to contain Atoms, then Atom must be on MoleculeBase's template statement argument list:

```
template <class Atom> class MoleculeBase { ...
```

If class AtomBase needs to offer an access function returning type Molecule, then Molecule must be on AtomBase's template statement argument list:

```
template <class Molecule> class AtomBase { ...
```

This is all fine unless MoleculeBase needs to offer friendship to AtomBase or to AtomBase member functions. In order to make the friendship declaration, the template statement arguments for AtomBase need to be known to MoleculeBase. The only way to supply them is via MoleculeBase's template statement argument list. This means that Molecule needs to be on MoleculeBase's template statement argument list:

```
template <class Atom, class Molecule> class MoleculeBase {
        friend class AtomBase<Molecule>;
        ...
```

It also means that any future additions to AtomBase that affect AtomBase's template statement (for example, support for Rings) also have to be added to MoleculeBase's template statement whether or not those changes are relevant to MoleculeBase, i.e. changes to AtomBase are no longer encapsulated.

The upshot is that every template class may wind up needing every other template class on its argument list. This destroys the advantage of encapsulation since adding a new class to the library, such as Ring, requires modification of every other class in the library, not just those classes that use the new class.

Since encapsulation is broken, the library is no longer extensible; two different developers can no longer extend the capabilities of the library in two different (but separate and distinct) ways since the changes propagate throughout the entire library and thereby make the libraries incompatible with one another. This means that re-use of functionality is destroyed.

True Modularity Cannot Be Achieved

Modularity is an important feature of a class library. A modular class library supports the following features:

1. A library class can extend the functionality of another simpler library class, and an application can use either the extended class or the simpler class, according to its needs.
2. Two different library classes can extend the functionality of the same simpler library class in distinct ways, and an application can make use of the simpler class, either extended class, or both extended classes.
3. The same functionality may be provided by multiple library classes using different implementations. The application program should be able to choose the appropriate implementation freely.
4. If a library class extends the functionality of a simpler library class, it should be able to extend any library class that provides the simpler class' functionality.

It is impossible to support all the above in a C++ class library in a general way.

Point #4 implies that an extending class must inherit from the simpler class it is extending, since the extending class cannot know all the functionality provided by the simpler class (it can only know the subset that it needs) and that therefore it cannot use a mechanism such as having the simpler class as member data and providing wrapper calls to the simpler class functionality.

Since the functionality extension mechanism must be inheritance, points #3 and #4 both imply that the simpler class must be provided as a template argument to the extending class, since the extending class must be able to extend any class providing the necessary base functionality.

This ultimately leads to an unresolvable conflict due to the way constructors work. The extending class cannot know what specific base class it is extending. It may be extending a minimal base class that provides only the functionality it needs, or it may be extending a more extensive base class that *requires additional constructor arguments*. There is no way that the extending class, in its constructor, can invoke the base class constructor with the proper arguments.

Genlib specifically supports all four of the above requirements for a modular class library.

Daunting Complexity

It has been our experience that the use of template base classes to construct a class library produces a library that is so complex that it is hard for the application developer to use and extremely difficult for the library developer to extend and maintain.

Template classes in and of themselves are difficult to use and maintain and also produce lengthy compile times, but, in addition, in order to support true modularity there are additional complexities that have to be introduced: virtual base classes and templated inheritance lists.

Virtual Base Classes

Virtual base classes are used when two or more classes are derived from the same base class, and those derived classes are to be used together as base classes (via multiple inheritance) for a further derived class. In order to avoid having multiple instances of the "grandparent" base class, the parent classes have to be declared virtual on the inheritance list.

In a modular class library essentially every class will use virtual inheritance because inheritance lists will be supplied as template arguments (see next section) and therefore it will impossible to know beforehand if parent classes share a common ancestor.

The ugly part about using virtual base classes is that C++ mandates that all such classes' constructors must be invoked from the constructor for the "most derived" class (*i.e.*, the final application class), rather than from the constructor for the class that derives directly from the virtual base class. Since every class in the library needs to be virtual for reasons discussed above, this means that an application class has to invoke the constructor for every class it inherits directly or indirectly from. This pushes internals of the class library (how a class wants to invoke the constructor of its parent class) into the arena of the application programmer. This forces the application programmer to learn unnecessary details of the internals of the library — and those details may be complex, *e.g.*, a library class may want to invoke its parent's class constructor in different ways depending on which of its own constructors was used or depending on what argument values were supplied to its own constructor.

Templatized Inheritance Lists

A class library may support several different implementations of the same class functionality. This may be because of performance tradeoffs in various application environments, or because some core functionality offered by one class is also offered by other classes but extended in various ways. This implies that a library class that needs to inherit some particular functionality may not know what base class it is most advantageous to inherit from, given the final application's needs. The remedy to this is to have the inheritance list supplied as template arguments.

There are problems with using template arguments, though. For example if functionality X is provided by class A, and functionality Y is provided by class B, and class C provides both X and Y, then if a class needs to inherit functionalities X and Y it needs *two* template arguments if it uses classes A and B, but only *one* if it uses class C. The only workaround for this is to provide a wrapper class D that combines A and B so that inheritance would come from either C or D. It would be nice if you could simply inherit from either {A and B} or {C twice} (since C is a virtual base class, see above), but unfortunately the latter is illegal.

This leaves using a wrapper class as the only workaround. Clearly, scalability is a major problem of this workaround for a class library of any size. For example, if there are three classes (A1, A2, A3) providing functionality X and two classes (B1, B2) providing functionality Y, then there has to be six wrapper classes for the various possible combinations. Of course, the wrapper class itself could be a template class, but that way lies madness.

Appendix 2: *Genlib* input file for **gencollagen**

The *genlib* input file for the program **gencollagen** is presented below in its entirety. A discussion of the file's contents follows the file listing. The numbers preceding each line are not part of the file and are for reference during the discussion section.

```
1   implementation Molecule/default;
2   implementation Molecule/PDBio_default;

3   includefile <stdio.h> in Atom;

4   members Atom {
5           bool    loaded_;
6   public:
7           bool    isLoaded() const { return loaded_; }
8           void    setCoord(float x, float y, float z);
9           void    setCoord(const MolPos &pos);
10          void    dump(FILE *fp) const;
11  private:
12          void    setLoaded(bool flag) { loaded_ = flag; }
13  }

14  members Residue {
15          Residue *prev_;
16          float   phi_, psi_;
17  public:
18          Residue *prev() const { return prev_; }
19          float   phi() const { return phi_; }
20          float   psi() const { return psi_; }
21          void    setPrev(Residue *prev) { prev_ = prev; }
22          void    setPhi(float angle) { phi_ = angle; }
23          void    setPsi(float angle) { psi_ = angle; }
24  }

25  constructor Atom {
26          loaded_ = false;
27  }

28  constructor Residue {
29          prev_ = NULL;
30          phi_ = psi_ = 0;
31   }
```

Line 1 causes *genlib* to include the default implementations of the classes Molecule, Residue, Atom, Bond, Coord, and CoordSet from the Molecule toolkit. Line 2 includes the default implementation of the PDBio class, which can be used to read PDB files into (or write PDB files from) the aforementioned classes.

Line 3 causes <*stdio.h*> to be #included in the Atom class interface file (*Atom.h*).

Lines 4 through 13 specify additions to the Atom class interface. Line 12 adds a private instance variable. Lines 7 through 10 specify four additional public member functions. Line 12 adds a private member function. Note that all the added member functions are inline except for the one on line 10. That function is only three lines long and could have been provided inline, but the **gencollagen** author decided to provide it in a separate C++ source file. Alternatively, the entire non-inline function could have been provided in the *genlib* input file, embedded in a "code Atom {...}" construct.

Lines 14 through 24 add instance variables and member functions to class Residue in the same manner that lines 4 through 13 did so for class Atom.

Lines 25 through 27 add code to the Atom class constructor. Lines 28 through 31 do the same for the Residue class constructor.

This **gencollagen** configuration file, when processed using *genlib*, results in approximately 1500 lines of C++ code.

References

1. For more information on mmCIF (macromolecular Crystallographic Information File), please see the web site at http://ndbserver.nibh.go.jp/NDB/mmcif/.

2. For more information on CEX (Chemical Exchange Format), please see the web site at: http://www.cgl.ucsf.edu/cex.

3. IUPAC-IUC Commission on Biochemical Nomenclature, “Abbreviations and Symbols for the Description of the Conformation of Polypeptide Chains. Tentative Rules (1969)”, *J. Biol. Chem.* (1970), **245**, 6489.

4. F.C. Bernstein, T.F. Koetzle, G.J.B. Williams, E.F. Meyer, Jr., M.D. Brice, J.R. Rodgers, O. Kennard, T. Shimanouchi and M. Tasumi, "The Protein Data Bank: A Computer-based Archival File for Macromolecular Structures", *J. Mol. Biol*. (1977) **112**, 535-542.

5. E.F. Abola, F.C. Bernstein, S.H. Bryant, T.F. Koetzle and J. Weng, "Protein Data Bank" in *Crystallographic Databases - Information Content, Software Systems, Scientific Applications*, F.H. Allen, G. Bergerhoff and R. Sievers, eds., Data Commission of the International Union of Crystallography, Bonn/Cambridge/Chester, (1987) 107-132.

6. G.S. Couch, E.F. Pettersen, C.C. Huang and T.E. Ferrin, "Annotating PDB Files with Scene Information," *J. Mol. Graph*. (1995) **13**,153-158.

8.W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz Jr., D.M. Ferguson, D.C. Spellmeyer, T. Fox, J.W. Caldwell and P.A. Kollman, ‘‘A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules,’’ *JACS* (1995), **117**, 5179-5197.

9. M.H. Miller, G. Nemethy and H.A. Scheraga, ‘‘Calculations of the Structures of Collagen Models. Role of Interchain Interactions in Determining the Triple-Helical Coiled-Coil Conformation. 2. Poly(glycyl-prolyl-hydroxyprolyl),’’ *Macromolecules* (1980), **13**, 470-478.

10. H. Sugeta and T. Miyazawa, ‘‘General Method for Calculating Helical Parameters of Polymer Chains from Bond Lengths, Bond Angles, and Internal-Rotation Angles,’’ *Biopolymers* (1967), **5**, 673-679.
11. R.L. Dunbrack Jr. and M. Karplus, ‘‘Backbone-dependent Rotamer Library for Proteins Application to Side-chain Prediction,’’ *J. Mol. Biol.* (1993), **230**, 543-574.
12. J.W. Ponder and F.M. Richards, ‘‘Tertiary templates for proteins. Use of packing criteria in the enumeration of allowed sequences for different structural classes,’’ *J. Mol. Biol.* (1987), **193**, 775-791.
13. For more information on Democritos, please see the web site at http://www.seqnet.dl.ac.uk/CBMT/democ/HOME.html.
14. For more information on PDBlib, please see the web site at http://www.sdsc.edu /pb/pdblib/pdblib.html.
15. For more information on BTL (Bioinformatics Template Library), please see the web site at http://www.cryst.bbk.ac.uk/classlib/.