

RE-USABLE COMPONENTS FOR STRUCTURAL BIOINFORMATICS.

Sophie I. Coon, Michel F. Sanner, Arthur J. Olson

The Molecular Graphics Laboratory. The Scripps Research Institute, La Jolla, CA.

Abstract:

One of the challenges in biocomputing is to enable the efficient use of a wide variety of rapidly evolving computational methods to simulate, analyze and understand the complex interactions of molecular systems. Our laboratory investigates several areas including molecular visualization, protein-ligand docking, protein-protein docking, molecular surfaces and the derivation of phenomenological potentials. In this paper, we present how we have used Python to develop independent components to deal with different aspects of structural bioinformatics. We contrast this “*language-centric*” approach with an “*application-centric*” approach. We will describe the following packages: MolKit: to read, write, represent and manipulate molecules; DejaVu: a 3D geometry visualization component; ViewerFramework: a component providing support for building dynamically extensible visualization applications. We then describe how these components have been combined together to create extensible applications such as the Python Molecule Viewer (PMV) and the AutoDock Toolkit (ADT).

I - The Challenge:

The number of programs available to compute molecular properties and/or simulate their behavior is large and growing rapidly. Not only are there many different computational techniques (e.g., molecular dynamics (MD), conformational analysis, quantum mechanics, distance geometry, docking methods, ab-initio methods ...) but they also come in many flavors and variations, using different force fields, search techniques and algorithmic details (e.g., continuous space vs. discrete, Cartesian vs. torsional). Even though some have a clear advantage over others for a specific problem, this is not the general case and one may try to apply several methods to determine which works best for a given problem. It is sometimes necessary to modify methods or use several in combination to obtain the desired results. For most of these computational methods, visualization plays a key role for setting initial parameters, analyzing results and even steering the computation itself. Here again, there is a plethora of tools available such as Rasmol [1], Midas+ [2], Insight (MSI) [3], Sybyl (Tripos) [4], AVS (Advanced Visualization Technologies Inc.) [5], data Explorer (IBM) [6] and VMD [7] each with its own strengths and weaknesses. Enabling the efficient use and inter-operation of all these computational methods is not a simple task since most

of these programs have not been designed to inter-operate. Moreover, these computational methods are often developed in research laboratories and tend to evolve rapidly. Very often the only interface is a more-or-less well-defined file format. For these reasons, it is difficult to combine these computational methods in an efficient way. Re-using any part of these codes is often very difficult. Even the comparison of similar methods requires a considerable amount of work to "port" a given scientific problem from one program to another. This suppresses creativity and provides only the most basic level of tool interoperability.

II - "Traditional" Approach:

When confronted with this situation, the first solution that comes to mind is to connect programs using filters, enabling the output of one program to be used as the input of another one. However, this approach has serious limitations. First of all, the level of interoperability is very low, i.e. the inter-operation is only possible at the program level rather than at the functional level. For instance, substituting one type of electrostatic calculation for another in a Molecular Dynamics (MD) code would typically require substantial coding as well as access to the source code and a good understanding of the data structures used by the program. Another weakness of this model comes from its complete lack of code re-use. Indeed, each program will re-implement functionality, such as parsing molecular data files, leading inevitably to new bugs and peculiarities. Finally interfacing a new method to a pool of n methods requires writing n filters. This quadratic behavior in the number of required filters prevents this approach from scaling beyond a small number of methods.

III - "Application-Centric" Approach:

Most programs are written to provide an answer to a specific question, with little or no thought given to the basic and independent sub-tasks that need to be carried out to get the answer. This leads to monolithic programs from which it is hard to isolate and extract code for re-use in another program. For instance, GRASP [8] is a molecular visualization and analysis program widely used by the biocomputing community. This program is particularly useful for the display and manipulation of the surfaces of molecules and their electrostatic properties. Such a program must carry out the following sub-tasks: 1) read the molecules, 2) compute a molecular surface, 3) compute an electrostatic potential, and 4) render the surface with the electrostatic potential mapped onto it. Even though these tasks are well defined, independent and shared by many other molecular-modeling programs, re-using the code corresponding to any of these tasks or substituting it by another implementation is often very challenging because of dependencies and assumptions made by the programmer. For instance, one might choose to pass the molecular data structure created by the molecule reader as the input for the second component which computes the surface. This creates an unnecessary dependency between these two components since the

calculation of the surface only requires atomic centers and radii.

The problem of code re-use and inter-operability has been addressed in programs such as AVS or Data Explorer (DX). These programs provide an environment in which components (also called modules) can be connected to create a computational pipeline. They comprise a large number of processing modules for a wide variety of operations such as: data input, image processing, surface and volume rendering, etc. These modules can be linked together graphically using a network editor to create a processing stream for a particular visualization or computation. They offer a mechanism for adding custom designed modules allowing the user to extend the environment with new computational methods. This approach has several benefits. First, it promotes code re-use and perhaps more importantly, it forces low-level developers (i.e. programmers who add new modules) to think in terms of *‘functionality to be added to an environment’* rather than *‘a program to carry out a task’*. In addition, the visual programming paradigm offered by these environments creates a new class of users who can combine existing modules to create new computational/visualization pipeline without having to be expert, low-level programmers. However, this approach has limitations too. The main one is that applications are *“self-centric”*. By *“self-centric”* we mean that they are in charge of what happens and when it happens, they impose a number of choices such as data structure, data types, and Application Program Interfaces (APIs) to the developers. In other words they define a framework in which new components can be developed. These problems are often exacerbated by the lack of scripting capability within these frameworks. Recently, environment such as VTK (Visualization ToolKit) [9,10] and MOE (Molecular Operating Environment) [11] have appeared. These environments have bindings for interpreted languages that add a great deal of flexibility and power. Yet, they have been designed as frameworks to be extended. In other words, they too have a self-centric philosophy, imposing themselves as *“the”* environment in which to develop applications.

In the following section we will define and discuss an alternative approach to modular software development which we call the *“language-centric”* approach. We will explain how this approach is better suited to solve our specific software development problems and briefly discuss our choice of Python [12,13,14] as a developing environment. This section will be followed by the presentation of some of the Python packages [15] we have developed to deal with different aspects of structural bioinformatics. We will conclude with lessons learned and a brief discussion of the suitability of Python for our software development approach.

IV - “Language-Centric” Approach: Python as a Development Environment:

From the AVS and DX approaches we have learned the importance of modular development and of rigorous compartmentalization of tasks: reading a molecule from a file has nothing to do with the way it is graphically represented and therefore these components

should be written independently of each other. Even though this requires a little more work and thought, it guarantees that the components will have much greater re-use in the long term. This idea of encapsulation is strongly supported by modern object-oriented programming languages. However, even while using object-oriented programming techniques, it is easy to create a large hierarchy of inter-dependent objects that are supposed to be used with a pre-defined context or framework. By choosing a high-level, interactive scripting language as a framework, we can minimize the constraints imposed by the framework (i.e. the choice of that language) while having a full fledged programming language to combine components at high level. This is what we call the "*language-centric*" approach. Components are developed as independent extensions of this high level language. They can be used to carry out a specific task with the only constraint being that they are used from within the chosen language. This might sound trivial, yet most programs, even those built around interpreted languages, do not have this property.

In this approach the high level language serves as a glue to combine independent components to rapidly create specialized applications. This approach provides excellent level of code re-usability, which in turn greatly reduces the time required to develop new applications. Having a set of basic components that are used in multiple applications facilitates code maintenance and increases code robustness.

We chose Python to be our developing environment for a number of reasons: its concise and almost pseudo-code like syntax, its modularity; its object oriented design, its profiling, debugging, reflection, introspection and self documenting capabilities and the availability of the Numeric extension [16] allowing the efficient storage and manipulation of large amounts of numerical data. Because of these reasons, Python is better suited than other programming languages to be an extension language for developing complete components.

We would like to emphasize again that the goal of our approach is to create independent modules which are constrained only by the language, rather than using the language to create a new framework with its own set of limitations and constraints. We then provide "software legos" that can be used in someone else's favorite framework.

V - Re-usable and Independent Components:

Our laboratory is interested in the development of novel computational technologies and application of this technology to the analysis and understanding of complex systems in structural molecular biology. Over the last couple of years, we have developed a number of Python packages to deal with many aspects of our daily work. The Python extensions described below can be combined together in high-level applications such as PMV (Python Molecule Viewer) or ADT (AutoDock Toolkit) or used independently to help us solve such problems. However, some of these components like DejaVu are more general and have already been re-used in other fields.

A - MolKit:

MolKit is a package developed in pure Python for reading, writing and building dynamically extensible tree-like hierarchical representations or molecular data-structures reflecting the internal structure of molecules.

To build a tree-like structure we have implemented a `TreeNode` object. Such a node can adopt children, which have to be instances of either a `TreeNode` or a class derived from it. Each node knows among other things, its parent in the tree, the root of the tree, which is the node that has no parent, and the type of its children. A node has a name that is unique amongst its sibling nodes. This name can also be used to build a unique name for each node in the tree. The `TreeNode` class supplies methods to generate such a unique name as well as methods to find a node from a name.

Such a tree can be queried for nodes fulfilling a given condition. The result is an instance of a `TreeNodeSet`. A `TreeNodeSet` represents a set of `TreeNode` object. The `TreeNodeSet` class implements Boolean operations as well as methods for string representation of the set. In addition, this class allows the easy access to attributes of the objects in the set (Fig 1). `TreeNodeSet` can be indexed and sliced just like Python lists, they also can be sorted using a depth-first traversal order.

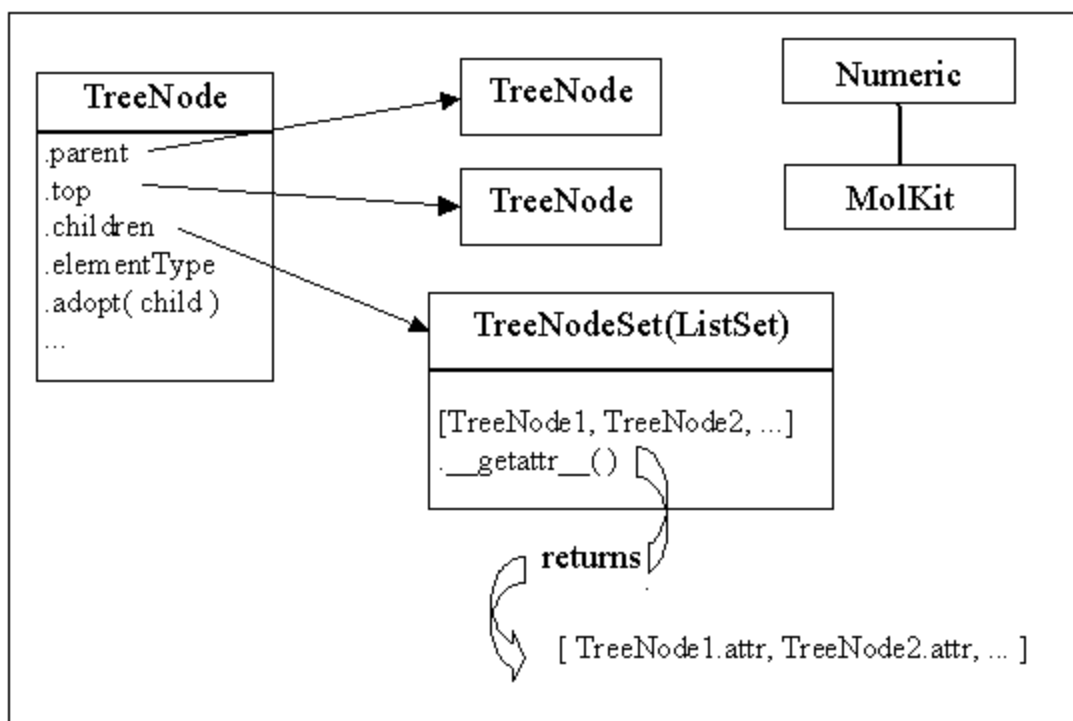


FIG 1: The `TreeNode` and `TreeNodeSet` classes.

These classes have been specialized to represent the internal structure of complex biological molecules. For instance, a protein molecule can be represented using the

following four level hierarchy: protein, chain, residue, atom. The number of level of a tree can be modified for instance we added a level between chain and residue to represent the secondary structure of a protein. MolKit also provides classes to parse molecular file formats such as PDB, Mol2, PQR and build these tree-like hierarchical representations. All nodes in that tree-structure can be extended dynamically. For instance one can add a property such as the radius of an atom called "A" by simply assigning the value to an attribute of this atom A: A.radius = 1.7. This structure which allows the user to extend dynamically any node and perform all kinds of selections has proven to be very powerful and to provide a high level interface to the molecule.

The following code instantiates a Protein object and builds its representation for a PDB file. Several selection and manipulation illustrate some of the basic capabilities of these classes:

```
>>>from MolKit import Read

>>># Read is a function of the MolKit package taking a filename

>>># and returning a ProteinSet. It can take any file of the following

>>># types: PDB, Mol2, PDBQS, PQR and PDBQ.

>>>proteins = Read("./1crn.pdb")

>>>protein = proteins[0]

>>># Once the protein is loaded, we can operate on that structure.

>>># TreeNodeSet object behave like Python lists.

>>>residues8Through15 = protein.chains[0].residues[8:16]

>>>print residues8Through15

<ResidueSet instance> holding 8 Residue

>>>

>>>allAtoms = protein.chains.residues.atoms

>>># select atoms in all atoms with their name in list:

>>>bbnames = ['N', 'CA', 'C', 'O']

>>>backBone = allAtoms.get(lambda x, names=bbnames: x.name in names)

>>># perform boolean operations over the set:

>>>sideChains = allAtoms - backBone

>>>
```

```

>>># Add a new members on the fly to any object or set of objects.

>>># Here we add an attribute called 'bb' to all the atoms and set it to 0

>>>allAtoms.bb = 0

>>># and set it to 1 for backbone atoms and 0 for side chain atoms.

>>>backBone.bb = 1

>>>

```

B - DejaVu:

DejaVu is a package written in Python for the visualization of 3D geometry using the OpenGL library. It provides a set of classes describing objects such as a viewer, cameras, lights, clipping planes, color editor, track-ball, geometries, etc. The Viewer class is a fully functional visualization application providing control over a number of rendering parameters such as depthcueing, global anti-aliasing, arbitrary clipping planes, etc. An instance of a Viewer maintains a hierarchy of objects in which rendering attributes can be inherited. DejaVu also provides a number of standard geometries including lines, indexed-lines, indexed-polygons, triangle, quad-strips, spheres, cylinders and labels. This list can be extended by sub-classing the geometry base class Geom. DejaVu makes it very trivial to add visualization capabilities to any object developed in Python. While developing this component we were careful not to make any assumption about the nature of the objects that will be represented in this viewer and therefore DejaVu has no knowledge of molecules. This has made this component re-usable in a context much broader than molecular visualization and modeling. We have already witnessed its successful re-use by people modeling heart and brain structures.

Combining DejaVu and MolKit, we can easily display a protein as space filling (CPK) models in a DejaVu Viewer (Fig 2). In the example below, after creating a tree representation of the protein Crambin (1crn), we extract the coordinates and the radii of the atoms and use them to instantiate Sphere geometries that can be displayed in a DejaVu Viewer. Since we only pass basic types (sequences of floating points values) from MolKit to DejaVu the viewer doesn't require knowledge about molecules. Therefore it can be re-used in a much larger context.

```

>>>from MolKit import Read

>>># read a PDB File.

>>>proteins = Read('./1crn.pdb')

>>>protein = proteins[0]

>>>from MolKit.molecule import Atom

```

```
>>># getting all the atoms of the molecule.

>>>allAtoms = protein.findType(Atom)

>>># assign a default radii to each atom.

>>>protein.defaultRadii(united = 1)

>>># getting the atoms coordinates and radii

>>>coords = allAtoms.coords

>>>radii = allAtoms.radius

>>>

>>>from DejaVu.Viewer import Viewer

>>># Create an instance of a Viewer.

>>>vi = Viewer()

>>>from DejaVu.Spheres import Spheres

>>># Create Spheres

>>>spheres = Spheres('cpk', centers=coords, radii=radii,
...                  quality=10)

>>># add the spheres to the viewer

>>> vi.AddObject(spheres)
```

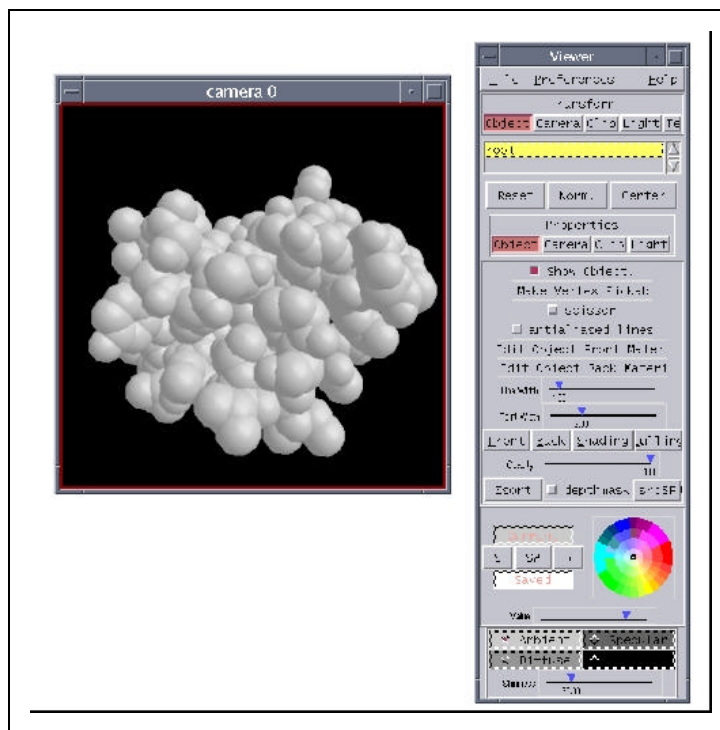


Fig2: CPK representation of the protein Crambin (1crn) in a DejaVu Viewer.

C - ViewerFramework:

The example given in the previous section illustrates how MolKit and DejaVu can be combined quite rapidly to achieve a simple visualization. However, end-user applications are much more complex. They require Graphical User Interfaces (GUI), structured commands, etc...

ViewerFramework provides support for building such visualization applications. It has been designed to have the following features: minimization the overhead of creating new commands and their associated GUI, dynamic configuration so that commands or set of commands called modules can be loaded dynamically from libraries, scripting capabilities enabled by the presence of a Python Shell, allowing access to the interpreter running the application, dual modes to call commands either through their GUI or through the Python Shell, a light weight command mechanism to add simple commands such as Macro, and finally support for logging commands allowing users to record and play back a session.

A ViewerFramework object is associated to a ViewerFrameworkGUI object that consists of a frame containing one or more menuBars, a canvas holding the 3D camera instance and a message box. Objects to be displayed in the ViewerFramework are stored in a list and are associated with GeomContainer objects holding a dictionary of geometries. In these dictionaries the keys are the geometry's name and the values the instances of

DejaVu geometries. Commands for an application derived from ViewerFramework can be developed by sub-classing the Command base class. The class CommandGUI allows to define GUI to be associated with a command.

Example:

```
# derive a new command.

from ViewerFramework.VFCommand import Command, CommandGUI

class ExitCommand(Command):

    def doit(self):

        import sys

        sys.exit()

# get a CommandGUI object

g = CommandGUI()

# add information to create a pull-down menu in a menubar called

# 'MoVi' under a menu-button called 'File' with a menu Command

# called 'Exit'. We also specify that we want a separator to

# appear above this entru in the menu.

g.addMenuCommand('MoVi', 'File','Exit',separatorAbove = 1)

# Add an instance of an ExitCommand with the alias 'myExit' to

# the viewer mv. This will automatically add the menu bar, the

# menu button (if necessary), the menu entry and bind the

# default callback function.

mv.addCommand(ExitCommand(), 'myExit', g)
```

The command is immediately operational and can be invoked through the pull down menu or using the Python Shell: `mv.myExit()` (Fig3).

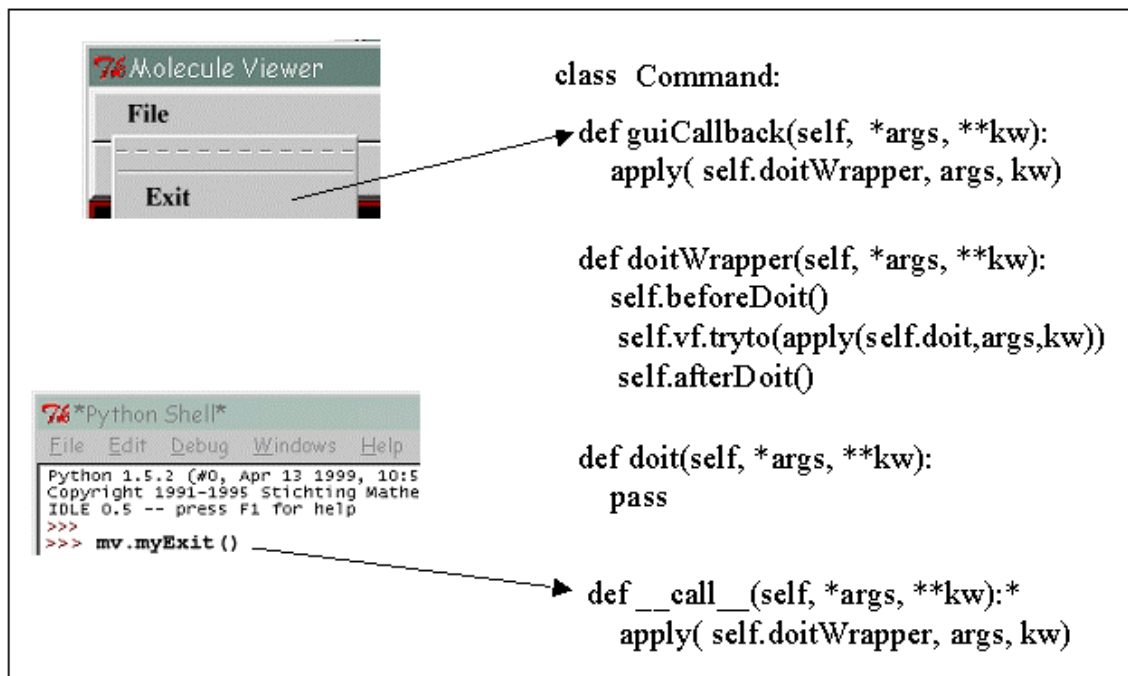


Fig3: Command base class: doit is a virtual method either called by the `__call__` or `guiCallback` method. These methods are the commands entrypoint through the PyShell or the GUI.

A number of related commands can be grouped into a module. A module is a .py files that defines a number of commands and provides a function called `initModule(viewer)` used to register the module with an instance of a viewer. When a module is added to a viewer the .py file is imported and the `initModule` function is executed. Usually this function instantiates a number of command objects and their associated `commandGUI` objects and adds them to the viewer.

D - PyBabel:

Babel is a program designed to interconnect a number of file formats. It is also capable of assigning hybridization, bond order and connectivity when these elements are not present in the input file. We have re-implemented in Python a subset of Babel-1.6 functions for assigning atomic types and bond orders, adding hydrogen atoms, computing Gasteiger charges, converting atom type and detecting rings and aromaticity. We find that not only is the code more portable but even performs better after replacing some $n*n$ algorithm from the C version by $n \log n$ versions. Another point that is important about this component is that we have kept it separate from the MolKit package. This choice was based on the idea that this component should be re-usable in other applications in which representation of a molecule might be quite different from the one implemented in MolKit.

This package has been proven to be very useful in the AutoDock Toolkit (ADT).

E- Other re-usable Computational Methods:

Using SWIG (Simple Wrapper Interface Generator) [17], we also have wrapped codes dealing with tasks such as: molecular surface computation, molecular mechanics and dynamics, electrostatic calculations, protein-protein docking and protein-ligand docking. Besides these “traditional” computational methods we have also wrapped a number of less conventional tools for molecular modeling such as a convex hull calculation tool, a rapid collision detection tools for molecular library, several mesh decimation algorithms and a general extrusion library. These modules are all platform dependent. They are also re-usable in a broader context. The functionality provided by most of these libraries has been made available as commands in our application PMV [18].

VI - From Building Blocs to Applications: PMV and ADT

A - PMV: Python Molecular Viewer:

The Python Molecular Viewer (PMV) is a general purpose viewer that can be integrated into any computational chemistry package available in Python. It relies on DejaVu for the 3-Dimensional visualization, ViewerFramework for the definition of individual commands and the GUI and MolKit for the representation of molecules (Fig4).

The Molecule Viewer inherits from the ViewerFramework the capability to dynamically import these commands as needed. In fact, all commands in that viewer have been developed based on this principle. This enables programmers to add features incrementally to the application, which is well suited for team development. In addition, this approach avoids the 'feature overload' problem, i.e. overloaded menus cluttered with commands that are irrelevant for the problem at hands. Customization files allow, among other things, to specify which commands should be loaded when the application starts. Using this feature, a number of custom applications can be created by simply writing different customization files, each loading different sets of commands.

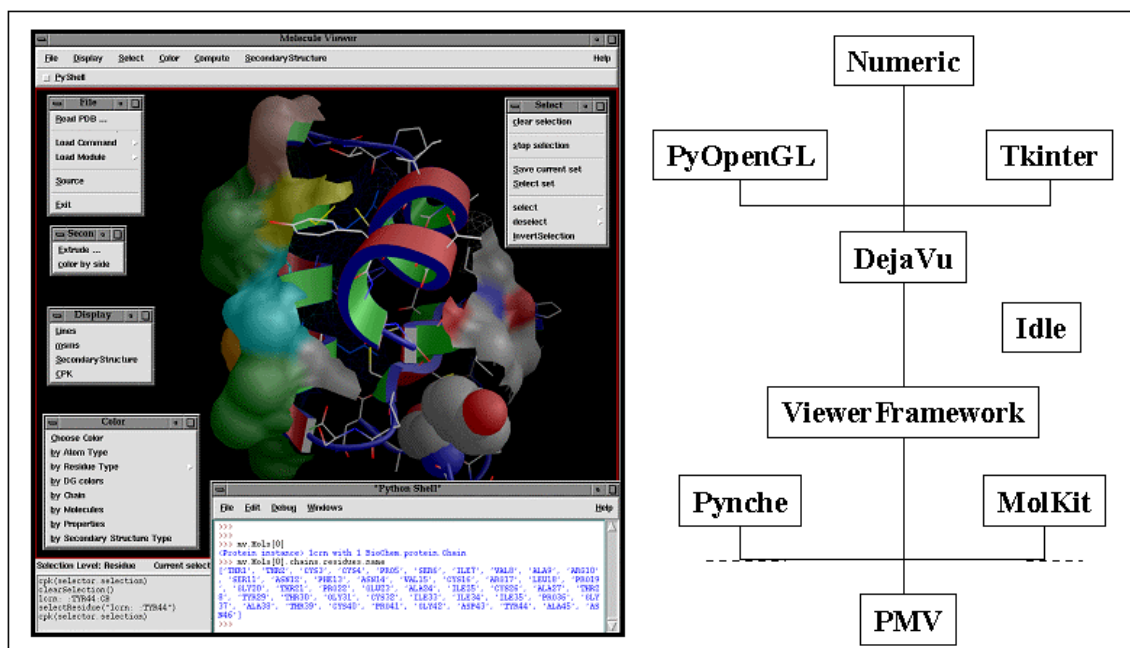


Fig 4: The Python Molecule Viewer (PMV) Architecture.

We have written a surface calculation command and a command to selectively display pieces of a surface corresponding to a subset of atoms. In fact in this viewer all geometries relate to the underlying molecules, therefore a subset of atoms can be used to partially display/undisplay, selectively label and locally color using different schemes any of the geometries. Alternatively, picking done on any geometry (lines, spheres, polygons, ...) is mapped back to the molecule. This application also provides a Python Shell. This Python aware type-in widget is the standard input and standard output of the scripts operating on the application. From the Python Shell, any command can be called interactively and scripts operating on the application can be written or sourced. Commands log themselves into a file as they are executed. This file can be saved, edited and played back to restore a previous session. This application runs unmodified on any platform that has a Python interpreter with the following extensions: Numeric, Tkinter/Togl and OpenGL. Maybe the most interesting feature of this application is that creating the application is actually done by writing commands, which is made easy by the built-in support provided by the ViewerFramework.

B - ADT: The AutoDock ToolKit.

The program AutoDock was developed to provide an automated procedure for predicting the interaction of ligands with biomacromolecular targets. The motivation for this work arises from problems in the design of bio-active compounds and in particular the field of computer-aided drug design. AutoDock has been shown to be a powerful approach to the problem of flexible docking. It combines configurational exploration of flexible ligands

with rapid energy evaluation using grid-based molecular affinity potential. The docking simulation can be carried out using several different search methods involving many possible parametric refinements. Preparation of the molecules also depends on various sets of parameters.

In response to the widespread use of AutoDock we developed a set of commands written for PMV to specifically address the problem of preparing the input files and launching and analyzing the results of a receptor-flexible ligand docking calculation using the AutoDock program. ADT consists of five modules: AutoTors; to format the small protein, AutoGpf: to create parameters file for Autogrid specifying grid types, location and dimension, AutoDpf: to create parameter file for AutoDock specifying molecules, grids, algorithm to use etc..., AutoStart: to start AutoGrid and AutoDock jobs and to manage them, AutoAnalyze: to analyze results of AutoDock jobs by looking at good docking and comparing them.

ADT first guides the user through the preparation of the molecule files (adding hydrogen atoms and charges and defining flexible regions) and the parameter files and then facilitates the execution of AutoGrid and AutoDock commands. The user can then choose how and where to launch the jobs. The macromolecule, the initial conformation of the ligand molecule, the docked conformations of the ligand and the isocontours showing the atomic affinity grids used by AutoDock can be viewed in ADT. In addition, the functionality provided by ADT can be used in script mode which enables the automation of AutoDock for non-interactive high throughput applications.

Conclusion:

We have presented a "language-centric" software development strategy and have demonstrated an implementation of such an approach based on the Python programming language. We have shown that it is possible, in such an environment, to develop independent components that can be assembled to build complex applications while retaining the ability to re-use most of these components in complete different context. While experimenting with this approach we have witnessed a dramatic increase in our productivity as well as a high level code re-use. We have also encountered some shortcomings in Python. The most problematic one is Python's incapability of handling circular references for garbage collection. We certainly hope that the optional garbage collection that has been added in Python-2.0 will help with this problem.

The most important change for us has been to shift from writing programs or scripts to writing modules or components that are re-usable and independent. This concept is well known to the community of component based software developers but in a molecular research environment with limited resources for the design and implementation of software tools Python has been instrumental. It has placed these advanced software development techniques within our reach.

References:

- [1] RASMOL: <http://www.umass.edu/microbio/rasmol/>
- [2] MidasPlus: <http://www.cgl.ucsf.edu/midasplus.html>
- [3] Insight: <http://www.msi.com/life/products/insight/index.html>
- [4] Sybyl: <http://www.tripos.com/software/sybyl.html>
- [5] C. Upson *et al.* IEEE Comput. Graphics Appl. 9(4), 30-42 (1989)
- [6] The Data Explorer home page: <http://www.research.ibm.com/dx>
- [7] Visual Molecular Dynamics: <http://ks.uiuc.edu/Research/vmd>
- [8] GRASP: <http://trantor.bioc.columbia.edu/grasp/>
- [9] W. Shroedr, K. Martin, B. Lorensen. The Visualization ToolKit: An object-oriented approach to 3D Graphics. Prentice Hall (1997).
- [10] VTK: <http://www.kitware.com/vtk.html>
- [11] MOE: <http://www.chem.ac.ru/Chemistry/Soft/MOPERENV.en.html>
- [12] M. Lutz. Programming Python. O'Reilly & Assoc. (1996)
- [13] M. Lutz, D. Asher. Learning Python. O'Reilly & Assoc. (1999)
- [14] Python home page: <http://www.python.org>
- [15] MGL python packages: <http://www.scripps.edu/~sanner/Python/documentation.html>
- [16] Numeric: <http://numpy.sourceforge.net>
- [17] SWIG: <http://swig.org>
- [18] Michel F. Sanner. Python: A programming Language for Software Integration and Development. J.Mol. Graphics Mod. (Feb 1999) Vol 17, 57-61.