SAND2004-1411P

Icarus Developer's Guide

Richard P. Muller

<rmuller@sandia.gov>

Revision History
Revision 0.5 February 2004

Table of Contents

Introduction

What is Icarus?

Icarus is an Interface for Chemical/Crystal Analysis and Simulation. The basic concept of Icarus is a set of defined data structures and modules that act on those structures. File I/O modules fill the data structures and write them back out to disk. Visualization modules give a picture of what the data looks like. And Miscellaneous modules modify the data contained in the data structures.

Because the data structures are defined outside of the modules, the modules can interact through them. This makes it easier to add new functionality to the program without breaking other features. It also helps speed development

because developers can take advantage of the file I/O and visualization code already built into Icarus. They can also take advantage of the features found in the other miscellaneous modules already built into Icarus.

Copyright, Licensing, and Authorship

Icarus was written at the California Institute of Technology by Ryan Martin, who also wrote most of this developer's guide. Shannon Stewman also contributed code to this project. The code is currently being maintained by Rick Muller at Sandia National Laboratories.

Programming for Icarus

Icarus is written in a programming language called Python. Python is an interactive language like Lisp. This means that the compile step in languages like C and Fortran is taken care of automatically by the Python interpreter. So the normal development cycle of code-compile-test-repeat becomes code-test-repeat. Python is also an Object Oriented language. So it has classes, inheritance, and other OO paradigms built into the core of the language. It is also a clean language that is fun to program in.

The windowing interface currently being used by Icarus in an open source framework called GTK+. GTK+ can be compiled on many different OS', including Linux, Irix, and Windows. So, together with Python, it makes Icarus a highly portable program. Most platforms that have support for Python and GTK+ will be able to run Icarus.

The following chapters describe the data model used in Icarus as well as the different module types. Most people will find "Data Structures" and "Misc Modules" the most useful.

Data Structures

There are two main data structures in Icarus: View and Material. These are made up of sub-structures such as Geometry, Geometryview, Atom, and Atomview. Visualization modules operate on Views and Miscellaneous Modules operate on Materials. File I/O modules either return a Material or operate on one.

Material Objects

A material object contains all data that applies to one material. Multiple simulations are stored in a material, multiple geometries are stored in a simulation, multiple atoms are stored in a geometry and so on. The following box shows a hierarchy of the material model.

```
Material
```

```
Simulation List
   Simulation
      Geometry List
         Geometry
            Atom List
            Bond List
               .
               .
               .
```

This example shows how to get a list of all the atoms and print out their positions.

```
atom_list=material.get_atom_list()
for atom in atom_list:
   print atom.get_position()
```

This example shows how to get a list of all the bonds in the material and print out the order of each bond.

```
bond_list=material.get_bond_list()
for bond in bond_list:
   print bond.get_order()
```

The next few examples show how to drill down the object hierarchy in a material object. The first level below a material is a simulation. This holds all the data from a particular simulation, including a list of all the geometries from the simulation. Below is an example showing how to get the list of simulations and print out the number of geometries in each.

```
simulation_list=material.get_simulation_list()
for sim in simulation_list:
   print len(sim.get_geometry_list())
```

This example shows the use of "active" attributes. In this example, the active simulation points to the simulation in the simulation list that is currently active(i.e. being viewed).

```
simulation=material.get_active_simulation()
for geo in simulation.get_geometry_list():
   print geo.get_timestamp(), geo.get_total_energy()
```

This example shows how to get the atom and bond lists in an alternative fashion. The *get_atom_list* and *get_bond_list* function used in the examples above get the atom and bond lists from the active geometry.

```
simulation=material.get_active_simulation()
geometry=simulation.get_active_geometry()
atom_list=geometry.get_atom_list()
```

```
bond_list=geometry.get_bond_list()
```

View Objects

View objects encapsulate all the data needed to draw a material object and it's components. Attributes such as color and radius are stored in this object. The view object hierarchy mirrors the material object hierarchy.

```
Materialview
   Simulationview List
      Simulationview
         Geometryview List
            Geometryview
               Atomview List
               Bondview List
                  .
                  .
                  .
```

This example shows how to print out the colors of all the atoms.

```
atomview_list=view.get_materialview().get_atomview_list()
for atomview in atomview_list:
   print atomview.get_color()
```

This example shows how to get the camera and print out what it is pointing at.

```
camera=view.get_camera()
print camera.look_at()
```

Module Development

FileIO Modules

File I/O modules implement read and write functions for the different types of files used in computer simulations. A few common types are Biograf/Polygraf(.bgf, .xtl), XMol(.xyz), Brookhaven(.pdb), and Tripos(.mol2). There are also modules for loading the input and output files of different programs, such as Jaguar.

A file I/O module defines 2 variables and one or two functions. The two variables are *extensions* and *filetype*. The two functions are *load(filename)* and *save(filename, material)*.

Icarus will first check the extension of the file to be loaded or saved and look for a file I/O module with that extension. If multiple modules have the same extension, Icarus will pop up a dialog box asking the user to select the module by *filetype*.

A file I/O module doesn't have to define both functions. For instance, the Jaguar output file I/O module only defines a *load* function because it makes no sense to provide a *save* function. You could, in fact, leave out both the *load* and *save* functions and Icarus wouldn't complain, but that is kind of pointless. The *load* function takes a single argument, a filename, and returns a Material object. The *save* function takes both a filename and a Material object to save. It returns nothing.

```
extensions=[".ext"]
filetype="Example"

load(filename)          #Returns a Material object
save(filename, material)
```

An example of the IO module for the XMol XYZ format follows:

```
#xyz.py
#
# Author: Ryan Martin
# Date: 6/14/00
#
# structure of file io module
#   extensions=["file extension"] (e.g. xyz, bgf)
#   filetype="file's identifying type" (e.g. Biograf)
#   load(filename)  read file function
#   save(filename, material)  write file function

from Material import Material
from Atom import Atom
from Bond import Bond
import string
from os import path

#Set the filename extensions that this i/o module is used for.
extensions=["xyz"]
#Set a description of the filetype, e.g. Biograf, Jaguar Output.
filetype="??"

def load(fullfilename):
   #Split the input filename into its directory and filename components.
   filedir,filename=path.split(fullfilename)
   #Split it further into its name and extension.
   fileprefix,fileext=path.splitext(filename)
   #Create a new material with the name set to the file prefix.
   material=Material(fileprefix)
   #Open the file for reading.
   file=open(fullfilename, 'r')

   #Read the first line in the file.  This is the number of atoms.
   line = file.readline()
   #Split the line by white space.
   words = string.split(line)
   #The first(and only) element in the words array is the number of atoms.
```

```
    #Convert it to an integer.
    nat = int(words[0])

    #The next line in the file is the comment line.
    comment = file.readline()
    #Set a variable to use for numbering the atoms.
    num = 1
    #Loop over the number of atoms in the structure.
    for i in range(nat):
        #Read another line from the file.
        line = file.readline()
        #Split the line on white space.
        #Line format: atom_type x y z
        words=string.split(line)
        #Set a temporary variable to hold the type.
        type=words[0]
        #Create a temporary list to hold the position.
        pos = [float(words[1]),float(words[2]),float(words[3])]
        #Create a new atom and add it to the material created above.
        material.add_atom(Atom(num, pos, type))
        #Increment the atom number.
        num=num+1
    #Once all the atoms in the file have been read, return the material.
    return material


def save(filename, material):
    #Open the output file for writing.
    file=open(filename, 'w')
    #Create the number of atoms line.
    num_atoms="%i\n" % (len(material.get_atom_list()))
    #Print it to the file.
    file.write(num_atoms)
    #Write out an empty comment line.
    file.write("\n")        #comment line
    #Loop over all the atoms in the material.
    for atom in material.get_atom_list():
        #Get the current atom's position.
        pos=atom.get_position()
        #Create the atom line from the atom type and position.
        line="%s %f %f %f\n" % (atom.get_fftype(),pos[0],pos[1],pos[2])
        #Write the line to the file.
        file.write(line)
    #Close the file now that we are done with it.
    file.close()
```

Miscellaneous Modules

Miscellaneous modules are used to implement extra functionality in Icarus. They take the form of one or more windows used to interact with and modify the data stored in the data model. Because the infrastructure for miscellaneous modules makes no assumptions about what type of visual interface the module will have or what functionality it will implement, a miscellaneous module can do just about anything. Examples of some of the modules currently in Icarus

are a Jaguar control module, a trajectory animation module, and a module to calculate the geometric mean of a set of atoms. Most of the modules written for Icarus will take this form.

Miscellaneous modules have to implement two functions, *__init__(self)* and *run(self)*, and have the option of implementing four more, *initialize(self)*, *finalize(self)*, *update_window(self)*, and *update_selection(self)*. These six functions form the API for miscellaneous modules. Most modules will define the required *__init__(self)* and *run(self)* functions along with the optional *update_window(self)*.

```
class my_module(Misc_module):
   def __init__(self)
   def run(self)
   def update_window(self)
   def update_selection(self)
   def initialize(self)
   def finalize(self)
```

A simple example of a module to compute the geometric mean of a structure follows.

```
#GeoMean.py
#
# Authors: Georgios Zamanakos and Ryan Martin
# Date: 03/08/01
# Purpose: Calculate the geometric mean of a structure

from gtk import *
from Misc_module import Misc_module

#remember filename(minus .py) has to be the same as class name
#(Template.py==Template)
class GeoMean(Misc_module):
   def __init__(self):
      #call Misc_module.__init__ to set up some default behavior
      Misc_module.__init__(self)
      self.name="Geometric Mean"  #can be different from file/class name

   def run(self):
      #creates a gtk window called self.modulewindow
      Misc_module.run(self)
      #Create a virtual vertical box to contain the rest of the widgets
      self.module_vbox=GtkVBox()
      #Add the box to the main window
      self.modulewindow.add(self.module_vbox)
      #Create a label to identify the module
      self.label=GtkLabel("Geometric Mean")
      #Pack the label into the vertical box
      self.module_vbox.pack_start(self.label)
      #Create label that will show the system name
      self.system_name=GtkLabel("")
      #Pack the label into the vertical box
      self.module_vbox.pack_start(self.system_name)
      #Create label that will show the value of the geometric mean
```

```
        self.gm_label=GtkLabel("0.0 0.0 0.0")
        #Pack the label into the vertical box
        self.module_vbox.pack_start(self.gm_label)

        self.button_box=GtkHButtonBox()
        self.module_vbox.pack_start(self.button_box)
        self.runbutton=GtkButton("Run")
        self.runbutton.connect("clicked", self.Geometric_Mean)
        self.button_box.pack_start(self.runbutton)
        self.closebutton=GtkButton("Close")
        self.closebutton.connect("clicked", self.destroy_window)
        self.button_box.pack_start(self.closebutton)

        self.modulewindow.show_all()

    def update_window(self):
        pass

    def Geometric_Mean(self,event):
        #Get active(currently viewed) material object
        material=self.get_active_material()
        #Set the material name
        self.system_name.set_text(material.get_name())
        #Get the list of atom objects
        atom_list=material.get_atom_list()
        #Get the total number of atoms
        total_atoms=len(atom_list)
        #Initialize the variable that holds the geometric mean
        gm = [0,0,0]
        #Output some material information
        print ("Total Number of atoms in file %s is %d"
             % (material.get_name(),total_atoms))
        #Loop over all the atoms in the material
        for atom in atom_list:
            #Get the current atom's position
            pos=atom.get_position()
            #Add the position to the geometric mean
            gm[0]=gm[0]+pos[0]
            gm[1]=gm[1]+pos[1]
            gm[2]=gm[2]+pos[2]
        #Once all the positions have been added, divide by the total number
        #of atoms to get the mean
        gm[0]=gm[0]/total_atoms
        gm[1]=gm[1]/total_atoms
        gm[2]=gm[2]/total_atoms
        #Output the mean to the terminal window
        print gm
        #Display the mean in the graphic window
        self.gm_label.set_text(str("%f %f %f" % (gm[0],gm[1],gm[2])))
```

Visualization Modules

Visualization modules are used to change aspects of the currently viewed system, e.g. to hide all the hydroden atoms. They do this by operating on the view data structures.

Visualization modules have to implement two functions, *__init__(self)* and *run(self)*, and have the option of implementing four more, *initialize(self)*, *finalize(self)*, *update_window(self)*, and *update_selection(self)*. These six functions form the API for visualization modules. Most modules will define the required *__init__(self)* and *run(self)* functions along with the optional *update_window(self)*.

```
#Visualization_controls.py
#
# Author: Ryan Martin
# Date: 9/19/00

#Import the gtk object used below
from gtk import GtkHButtonBox, GtkButton

#Import the Visualization_module base class
from Visualization_module import Visualization_module
#Import some constants used for setting the view mode
from constants import *

#Derive a new module from the Visualization_module base class
class Visualization_controls(Visualization_module):
   #__init__ must be defined and must call the base classes __init__ function
   def __init__(self):
      #Call the base classes __init__function
      Visualization_module.__init__(self)
      #Set the name of the module
      self.name="Visualization Controls"

   #run also has to be defined
   def run(self):
      #Creates a toplevel window called modulewindow
      Visualization_module.run(self)

      #Create a button box to hold the action buttons
      self.button_box=GtkHButtonBox()
      #Add the button box to the module window
      self.modulewindow.add(self.button_box)
      #Create a "Hide Hydrogens" button
      self.hide_h_button=GtkButton("Hide Hydrogens")
      #Clicking on the button now calls the hide_h function
      self.hide_h_button.connect("clicked", self.hide_h)
      #Pack the button into the button box
      self.button_box.pack_start(self.hide_h_button)
      #Create the "Show Hydrogens" button
      self.show_h_button=GtkButton("Show Hydrogens")
      #Connect it to the show_h function
      self.show_h_button.connect("clicked", self.show_h)
      #Pack the button into the box
      self.button_box.pack_start(self.show_h_button)
      #Create the "Close" button
      self.close_button=GtkButton("Close")
```

```
        #Connect it to the destroy_window function
        #This function is defined in the Visualization_module base class
        self.close_button.connect("clicked", self.destroy_window)
        #Add the button to the button box
        self.button_box.pack_start(self.close_button)

        #Tell the module window to display all the gtk elements it contains
        self.modulewindow.show_all()

    def hide_h(self, event):
        #Get the currently displayed view
        view=self.get_active_view()
        #Get the materialview from the view
        materialview=view.get_materialview()
        #Get the atomviewlist and bondviewlist from the materialview
        atomviewlist=materialview.get_atomview_list()
        bondviewlist=materialview.get_bondview_list()
        #Loop over all the atoms and make them invisible if they are hydrogens
        for atomview in atomviewlist:
            if atomview.get_atom().get_fftype()[0]=='H':
                atomview.set_mode(INVISIBLE)
        #Loop over all the bonds and make them invisible if they have a hydrogen
        for bondview in bondviewlist:
            if (bondview.get_bond().get_atom1().get_fftype()[0]=='H' or
                bondview.get_bond().get_atom2().get_fftype()[0]=='H'):
                bondview.set_mode(INVISIBLE)
        #Set the view as changed
        #This will cause the display window to update, removing the invisible
        #hydrogens
        view.set_changed()

    def show_h(self, event):
        #Get the currently displayed view
        view=self.get_active_view()
        #Get the materialview from the view
        materialview=view.get_materialview()
        #Get the atomviewlist and bondviewlist from the materialview
        atomviewlist=materialview.get_atomview_list()
        bondviewlist=materialview.get_bondview_list()
        #Loop over all the atoms and make them visible if they are hydrogens
        for atomview in atomviewlist:
            if atomview.get_atom().get_fftype()[0]=='H':
                atomview.set_mode(materialview.get_mode())
        #Loop over all the bonds and make them visible if they have a hydrogen
        for bondview in bondviewlist:
            if (bondview.get_bond().get_atom1().get_fftype()[0]=='H' or
                bondview.get_bond().get_atom2().get_fftype()[0]=='H'):
                bondview.set_mode(materialview.get_mode())
        #Set the view as changed
        #This will cause the display window to update, showing the previously
        #invisible hydrogens
        view.set_changed()
```

Program Internals

Since most functionality will be added to Icarus through modules, the core infrastructure deals mostly with loading and running modules, as well as with keeping track of Material and View objects. The following sections go into more depth about each of these modules.

CDR: Central Data Repository

The central data repository (CDR) holds materials and the views associated with them. You can access them using these functions.

```
get_material(num)
get_material_names()
get_material_num_name_pairs()
add_material(material) #Note--captures Material pointer
remove_material(material)

get_view(num)
get_view_names()
get_view_num_name_pairs()
create_view(material)
create_dependent_view(materialview)
remove_view(view) #Note--does not remove Material
```

FileIO Package

Section to be written.

Visualization Package

The visualization package loads and keeps track of all the visualization modules. The API for this package is:

```
load_modules(module_path)
load_module(filename)
unload_module_by_path(module_path)
unload_module(module)
reload_module(module)
reload_modules()
get_module_name_key_pairs()
run_module(module_name, view_control)
```

View Defaults

The view defaults package loads and keeps track of the default values for things such as atom color and radius by fftype. The API for these functions are:

```
load_color_dictionary(filename)
```

```
load_radius_dictionary(filename)
get_atom_color(atom_type)
get_atom_radius(atom_type)
```

View Control

View control objects connect a view window to the view/material that it is viewing. The next section describes its API.

```
class View_control:
   __init__(self)
   get_view_num_name_pairs(self)
   change_view(self, num)
   change_to_newest_view(self)
   get_active_view(self)
   get_active_view_ID(self)
   get_active_material(self)
   register(self, observer)
   notify(self)
```

Batch Interface

The batch job interface allows you to run Icarus in batch mode and have access to the data structures and file I/O features without using the GUI.

The following example loads a bgf file, prints out all the atom positions and force field types, sets all the positions to 0, and saves the file.

```
import Fileio, CDR

Fileio.load_material("ara.bgf")
material=CDR.get_material(0)
simulation=material.get_active_simulation()
geometry=simulation.get_active_geometry()
atom_list=geometry.get_atom_list()
for atom in atom_list:
   print atom.get_position(), atom.get_fftype()
   atom.set_position([0,0,0])
Fileio.save_material("ara_save.bgf", material)
```

Appendices

Material Classes

Material

```
Material:
   _name
   _simulation_dict
   _active_simulation
   _next_available_simulation_ID
   _observers

   get_name()
   get_next_sim_ID()
   add_simulation(simulation)
   remove_simulation(simulation)
   get_simulation_list()
   get_active_simulation()
   set_active_simulation(simulation)
   register(observer)
   notify()

   get_atom_list()
   get_atom(ID)
   add_atom(atom)
   remove_atom(atom)
   get_bond_list()
   get_bond(ID)
   add_bond(bond)
   remove_bond(bond)
   get_cell()
   get_residue_list()
   get_residue(ID)
   add_residue(residue)
   remove_residue(residue)
   get_molecule_list()
   get_molecule(ID)
   add_molecule(molecule)
   remove_molecule(molecule)
```

Simulation

```
Simulation:
   _ID
   _basis_function
   _hamiltonian
   _spin_multiplicity
   _geometry_dict{} (->geometry)
      key=timestep
   _active_geometry (->geometry)
   _residue_dict{} (->residue)
   _molecule_dict{} (->molecule)

   get_ID()
   set_ID(ID)
   get_basis_function()
```

```
   set_basis_function(basis_function)
   get_hamiltonian()
   set_hamiltonian(hamiltonian)
   get_spin_multiplicity()
   set_spin_multiplicity(spin_multiplicity)
   get_geometry_list()
   get_geometry(timestamp)
   get_copy_of_previous_geometry(timestamp)
   add_geometry(geometry)
   remove_geometry(geometry)
   get_active_geometry()
   set_active_geometry(geometry)
   increment_active_geometry()
   decrement_active_geometry()
   get_residue_list()
   get_sorted_residue_list()
   get_residue(ID)
   add_residue(residue)
   remove_residue(residue)
   get_molecule_list()
   get_sorted_molecule_list()
   get_molecule(ID)
   add_molecule(molecule)
   remove_molecule(molecule)

   get_atom_list()
   get_atom(ID)
   add_atom(atom)
   remove_atom(atom)
   get_bond_list()
   get_bond(ID)
   add_bond(bond)
   remove_bond(bond)
   get_cell()
```

Geometry

```
Geometry:
   _timestamp
   _temperature
   _total_energy
   _kinetic_energy
   _potential_energy
   _cell
   _atom_dict{} (->atom)
   _bond_dict{} (->bond)

   get_timestamp()
   get_temperature()
   set_temperature(temperature)
   get_total_energy()
   set_total_energy(energy)
```

```
   get_kinetic_energy()
   set_kinetic_energy(energy)
   get_potential_energy()
   set_potential_energy(energy)
   get_cell()
   set_cell(cell)
   get_atom_list()
   get_sorted_atom_list()
   get_atom(ID)
   add_atom(atom)
   remove_atom(atom)
   get_bond_list()
   get_sorted_bond_list()
   get_bond(ID)
   add_bond(bond)
   remove_bond(bond)
```

Cell

```
Cell:
   _a
   _b
   _c
   _alpha
   _beta
   _gamma
   _orlist (length=6)                    #Conversion matrix
      # orlist[0]=a*sin(beta)*sin(gamma_bar)
      # orlist[1]=a*sin(beta)*cos(gamma_bar)
      # orlist[2]=b*sin(alpha)
      # orlist[3]=a*sin(beta)
      # orlist[4]=b*sin(alpha)
      # orlist[5]=c
      # gamma_bar=(cos(gamma)-cos(alpha)*cos(beta))/(sin(alpha)*sin(beta))
   _cellcoord (length=8)                  #Corners of the cell box
      # Below are the xyz vectors of each value
      # cellCoord[0]=000 cellCoord[1]=001 cellCoord[2]=010 cellCoord[3]=011
      # cellCoord[4]=100 cellCoord[5]=101 cellCoord[6]=110 cellCoord[7]=111

   #Formula for converting from scale to real cell coords
   # x=scaled_x*orlist[0]
   # y=scaled_x*orlist[1]+scaled_y*orlist[2]
   # z=scaled_x*orlist[3]+scaled_y*orlist[4]+scaled_z*orlist[5]

   calculate_orlist()
   calculate_cell_coordinates()
   add_displacement(vector, displacement)
   set_a(a)
   set_b(b)
   set_c(c)
   set_alpha(alpha)
   set_beta(beta)
```

```
   set_gamma(gamma)
   get_a()
   get_b()
   get_c()
   get_alpha()
   get_beta()
   get_gamma()
   get_orlist()
   get_cell_coordinates()
```

Molecule

```
Molecule:
   _ID
   _atom_list[] (->atom)

   optional:
      _label
      _charge
      _radius_of_gyration
      _moment_of_inertia

   add_atom(atom)
   remove_atom(atom)
   set_ID(ID)
   set_label(label)
   calculate_charge()
   calculate_radius_of_gyration()
   calculate_moment_of_inertia()
   get_atom_list()
   get_ID()
   get_label()
   get_charge()
   get_radius_of_gyration()
   get_moment_of_inertia()
```

Residue

```
Residue:
   _ID
   _atom_list[] (->atom)

   optional:
      _label
      _charge
      _radius_of_gyration
      _moment_of_inertia

   add_atom(atom)
   remove_atom(atom)
```

```
   set_ID(ID)
   set_label(label)
   calculate_charge()
   calculate_radius_of_gyration()
   calculate_moment_of_inertia()
   get_atom_list()
   get_ID()
   get_label()
   get_charge()
   get_radius_of_gyration()
   get_moment_of_inertia()
```

Atom

```
Atom:
   _ID
   _position[x,y,z]
   _fftype

   optional:
      _label
      _charge
      _max_covalent_bonds
      _num_lone_pairs
      _bond_list[] (->bond)
      _force[3]
      _velocity[3]

   set_ID(ID)
   set_label(label)
   set_position(position)
   set_fftype(fftype)
   set_max_covalent_bonds(max_covalent_bonds)
   set_num_lone_pairs(num_lone_pairs)
   set_charge(charge)
   set_force(force)
   set_velocity(velocity)
   add_bond(bond)
   get_ID()
   get_label()
   get_position()
   get_fftype()
   get_max_covalent_bonds()
   get_num_lone_pairs()
   get_charge()
   get_force()
   get_velocity()
   get_bond_list()
```

Bond

```
Bond:
   _ID
   _atom1 (->)
   _atom2 (->)

   optional:
      _order
      _disp   (atom2 with respect to atom1)  at1-->at2

   set_ID(ID)
   set_atom1(atom)
   set_atom2(atom)
   set_order(order)
   set_disp(disp)
   get_ID()
   get_atom1()
   get_atom2()
   get_order()
   get_disp()
```

View Classes

View

```
View:
   _materialview (->)
   _camera
   _pixel_width
   _pixel_height
   _scale_factor
   _view_angle
   _changed

   set_camera(position, target, up_vector, right_vector)
   set_view_size(width, height)
   set_view_angle(angle)
   reset_changed()
   set_changed()
   get_materialview()
   get_camera()
   get_view_height()
   get_view_width()
   get_view_angle()
   get_name()
   has_change()
```

MaterialView

```
Materialview:
```

```
   _mode
   _material (->)
   _changed
   _active_simulationview

   get_mode()
   set_mode(mode)
   has_changed()
   set_changed()
   reset_changed()
   get_active_simulationview()
   get_material()
   update_material()
   get_atomview_list()
   get_bondview_list()
   get_cellview()
   get_name()
```

Simulationview

```
Simulationview:
   _mode
   _active_geometryview

   get_mode()
   set_mode(mode)
   get_active_geometryview()
   get_atomview_list()
   get_bondview_list()
   get_cellview()
```

Geometryview

```
Geometryview:
   _mode
   _atomview_dict
   _bondview_dict
   _cellview

   get_mode()
   set_mode(mode)
   add_atom(atom)
   add_bond(bond)
   add_cell(cell)
   add_atomview(atomview)
   remove_atomview(atomview)
   get_atomview(ID)
   get_atomview_list()
   add_bondview(bondview)
   remove_bondview(bondview)
```

```
   get_bondview(ID)
   get_bondview_list()
   update_halfbond_positions()
   set_cellview(cellview)
   get_cellview()
```

Cellview

```
Cellview:
   _cell (->)
   _mode

   set_mode(mode)
   get_mode()
   get_cell()
```

Atomview

```
Atomview:
   _atom (->)
   _color
   _radius
   _mode

   set_color(color)
   set_radius(radius)
   set_mode(mode)
   add_bondview(bondview)
   get_atom()
   get_color()
   get_radius()
   get_mode()
   get_bondview_list()
```

Bondview

```
Bondview:
   _bond (->)
   _halfbond_position1
   _halfbond_position2
   _radius
   _mode
   _atomview1 (->)
   _atomview2 (->)

   set_halfbond_position1(halfbond_position)
   set_halfbond_position2(halfbond_position)
   set_radius(radius)
```

```
set_mode(mode)
set_atomview1(atomview)
set_atomview2(atomview)
get_bond()
get_halfbond_position1()
get_halfbond_position2()
get_radius()
get_mode()
get_atomview1()
get_atomview2()
```