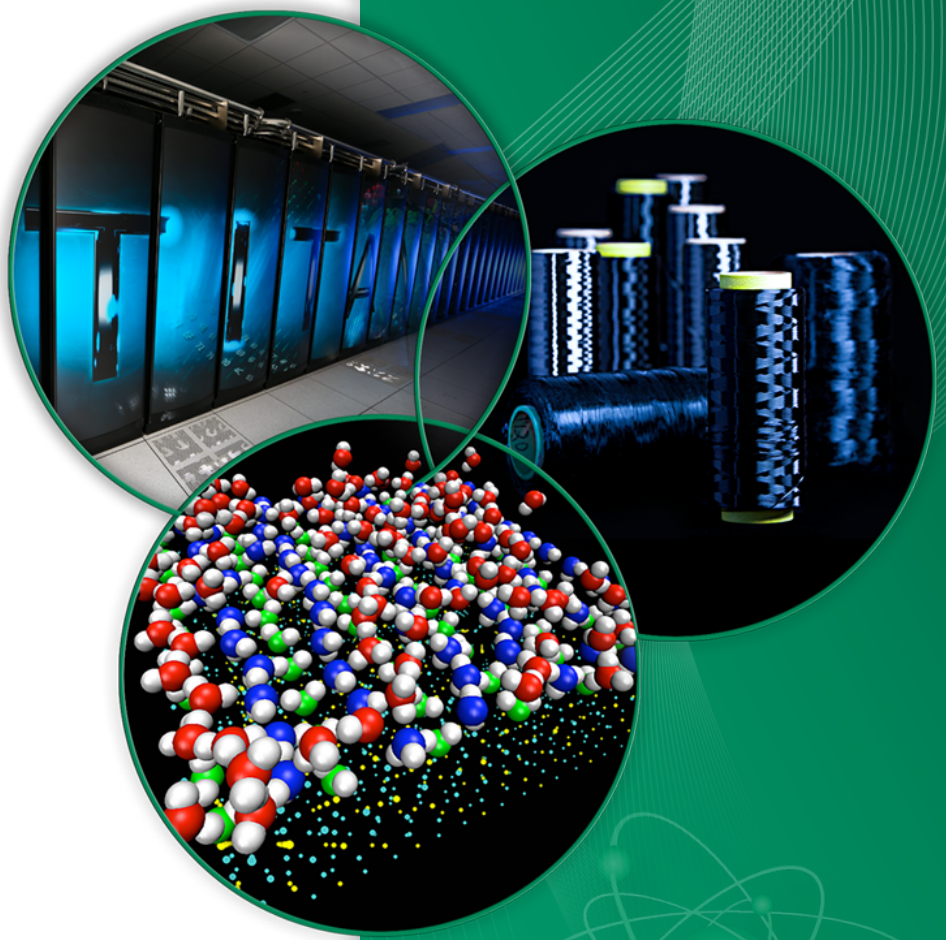
SasView Architecture Overview

SasView Code Camp

April 2-7, 2013

Mathieu Doucet



What are we talking about

- General information
  - Where to find things
  - Build servers
  - Build support
- Code organization
- Class diagrams
  - Application design
  - Loaders
  - Models
- Development tips

Where is everything?

- Web site: http://sasview.org/
- Build servers:
  - ORNL: https://builds.sns.gov/view/Other%20Software/view/SansView/
  - ISIS: http://download.mantidproject.org/jenkins/view/All/job/sasview_snowleopard_32bit/
- Code: http://sourceforge.net/projects/sasview/
- To find it all:

  http://danse.chem.utk.edu/daily_dev.html

- We need a better build server solution for Windows and OSX before the next release

ORNL build server

- Used for dev and release builds
- No longer supports Snow Leopard

S	W	Name	Last Success
		sansview_app_leopard_32bit	1 yr 0 mo (#29)
		sansview_app_lion_32bit	1 yr 1 mo (#8)
		sansview_app_snowleopard_32bit	3 mo 0 days (#671)
		sansview_code_doc	3 mo 0 days (#234)
		sansview_installer_windows7_32bit	13 days (#1181)
		sansview_leopard_32bit	1 yr 0 mo (#33)
		sansview_lion_32bit	1 yr 0 mo (#25)
		sansview_rhel6	11 hr (#817)
		sansview_snowleopard_32bit	3 mo 0 days (#785)
		sansview_windows7_32bit	13 days (#1195)

ISIS build server

- Meant to be used for Snow Leopard dev and release builds
- Still needs to be completely set up
- A detailed description of how to build on the Mac can be found here: http://sourceforge.net/apps/trac/sansviewproject/wiki/MacBuild

Build support

- Using the top-level `setup.py` is enough to install SasView and run it from the command line.
- Windows: `sansview/setup_exe.py` is used to generate a Windows executable.
- Mac: `sansview/setup_mac.py` is used to generate a Mac app.
- RHEL: The `build_tools/rpm` directory contains a script that will generate a spec file to build an rpm.

New developers: You don't need to compile an installer during development. Just run

```
python sansview.py
```

What is installed when running setup.py

This creates a script to start SasView.

```
340  # Set up SasView
341 setup(
342     name="sasview",
343     version = VERSION,
344     description = "SasView application",
345     author = "University of Tennessee",
346     author_email = "sansdanse@gmail.com",
347     url = "http://danse.chem.utk.edu",
348     license = "PSF",
349     keywords = "small-angle x-ray and neutron scattering analysis",
350     download_url = "https://sourceforge.net/projects/sansviewproject/files/",
351     package_dir = package_dir,
352     packages = packages,
353     package_data = package_data,
354     ext_modules = ext_modules,
355     install_requires = required,
356     zip_safe = False,
357     entry_points = {
358                     'console_scripts':[
359                                        "sasview = sans.sansview.sansview:run",
360                                        ]
361                     },
362     cmdclass = {'build_ext': build_ext_subclass }
363     )
364
```

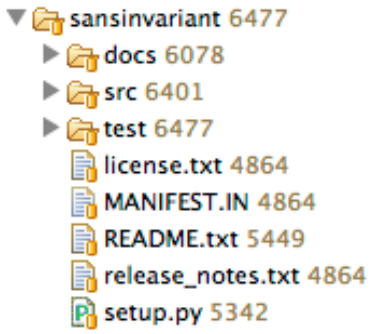
- This script is usually on the system path and it a convenient way to deploy the application on unix-like systems.

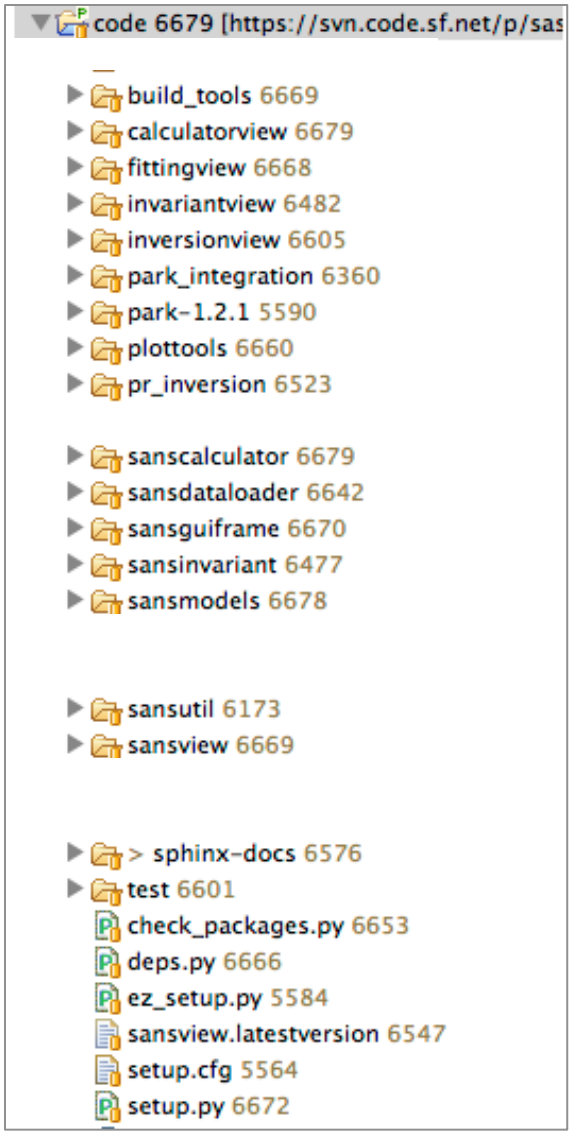
User data

- SasView writes to the user's home directory:
  - ~/sasview.log is the application log
  - ~/.sasview/config/custom_config.py contains customization settings
  - **~/.sasview/plugin_models contains your user models**

Code structure

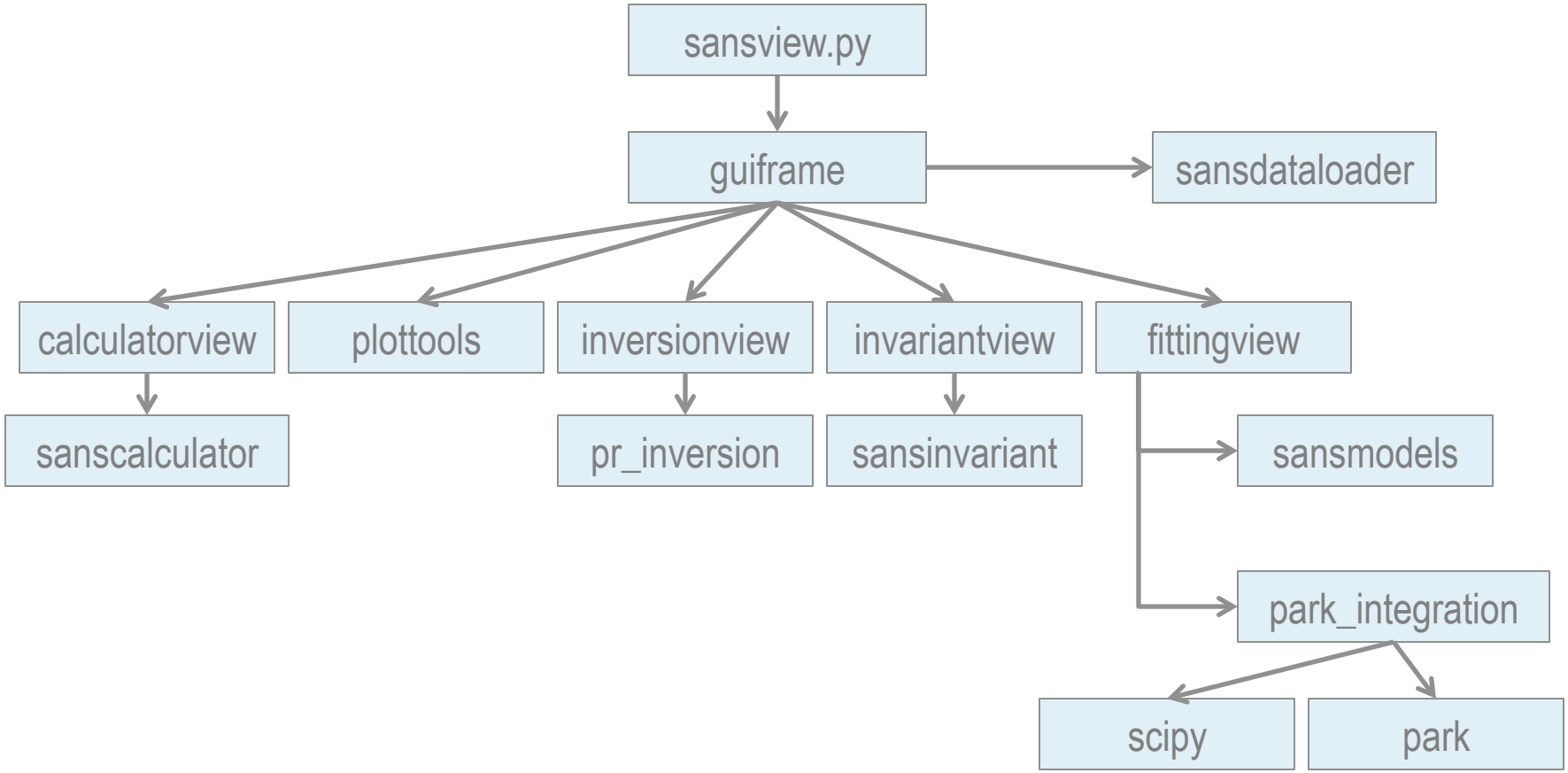
- Code is organized by packages
- The computations are done in a different module than the UI that presents them to the user
  - *sansinvariant* does the computations
  - *invariantview* is a UI plug-in
- Each module has a test directory for unit tests





Class diagram

- The following is a good source of class diagrams: http://danse.chem.utk.edu/diagrams.html



The SasView application

Initialization

‘Perspectives’

Welcome page

Build & start

```
class SasView():
    def __init__(self):
        self.gui = SasViewApp(0)
        self.gui.set_manager(self)

        # Fitting perspective
        import sans.perspectives.fitting as module
        fitting_plug = module.Plugin()
        self.gui.add_perspective(fitting_plug)

        # P(r) perspective
        import sans.perspectives.pr as module
        pr_plug = module.Plugin(standalone=False)
        self.gui.add_perspective(pr_plug)

        # Invariant perspective
        import sans.perspectives.invariant as module
        invariant_plug = module.Plugin(standalone=False)
        self.gui.add_perspective(invariant_plug)

        # Calculator perspective
        import sans.perspectives.calculator as module
        calculator_plug = module.Plugin(standalone=False)
        self.gui.add_perspective(calculator_plug)

        # Welcome page
        self.gui.set_welcome_panel(WelcomePanel)

        self.gui.build_gui()

        self.gui.MainLoop()
```

SasView plugin code

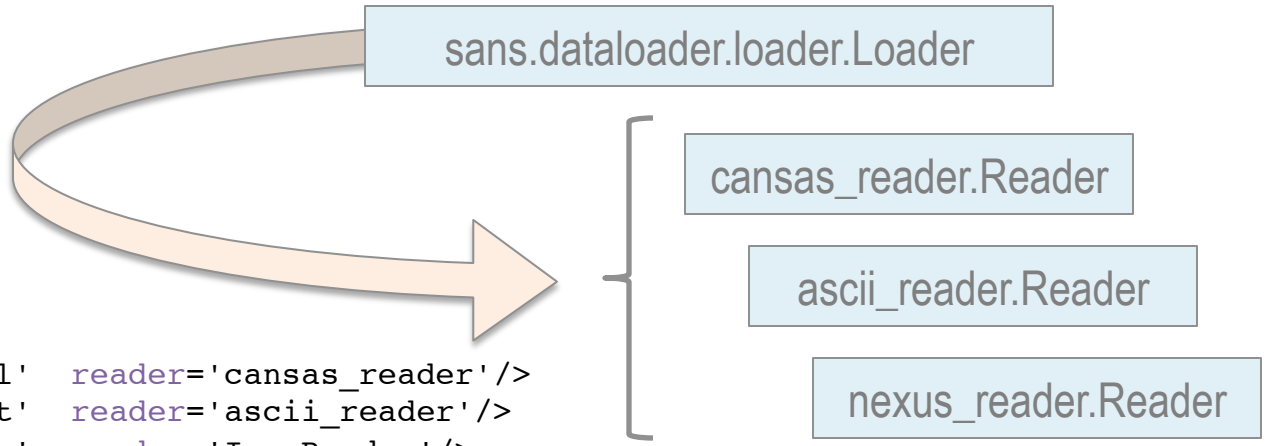
Fill those out to define panels and context menu items.

```
from sans.guiframe.plugin_base import PluginBase

class Plugin(PluginBase):

    def __init__(self, standalone=False): …

    def help(self, evt): …

    def get_panels(self, parent): …

    def get_context_menu(self, plotpanel=None): …

    def other_public_method(self): …

    def _other_private_method(self): …
```

See `inversionview/src/sans/perspectives/pr/pr.py` for an example.

Data loaders

- The loaders are registered with an XML settings file according to the file extension they deal with.




```
<?xml version="1.0"?>
<SansLoader version="1.0”>

    <FileType extension='.xml'  reader='cansas_reader'/>
    <FileType extension='.txt'  reader='ascii_reader'/>
    <FileType extension='.asc'  reader='IgorReader'/>
    <FileType extension='.dat'  reader='red2d_reader'/>
    <FileType extension='.tif'  reader='tiff_reader'/>
    <FileType extension='.abs'  reader='abs_reader'/>
    <FileType extension='.d1d'  reader='hfir1d_reader'/>
    <FileType extension='.sans' reader='danse_reader'/>
    <FileType extension='.nxs'  reader='nexus_reader'/>

</SansLoader>
```

A valid reader only needs a read() method that returns an instance of **sans.dataloader.data_info.Data1D**. A save() method can also be provided.

Loading files

Your Reader should be stateless because it's going to be re-used

```
from sans.dataloader.loader import Loader

# Loader is a singleton
loader = Loader()

# Loader tries the best loader for the given file
# When in trouble, it defaults to ASCII
data = loader.load("some_data_file.xml")

# You can programmatically associate a reader to
# a particular extension
loader.associate_file_reader(".mathieu", MatReader())

# If your Reader class has a save method…
loader.save("path_to_new_file", data, ".mathieu")
```

It's not a bad idea to look at the code in

sansdataloader/src/sans/dataloader/data_info.py

Basic python models

Only works with C++ models

```
class CylinderModel(BaseComponent):

def __init__(self):
    self.name = 'Cylinder'
    self.description = 'A cylinder model'
    self.details = {'radius': ['Angstrom', min, max],
                    'length': ['Angstrom', min, max],
                    'phi':    ['degrees', min, max]}
    self.orientation_params = ['phi']
    self.params = {'radius':20.0,
                   'length': 100.0,
                   'phi':0.0}
def run(self, x):
    return f(x)

def runXY(self, x):
    return f(x[0], x[1])

def evalDistribution(self, x):
    # x is either an array of x values (1D)
    return [ f(x_i) for x_i in x ]
    # or a list of lists… [x[], y[]]
    return [ f(x[0][i], x[1][i]) for i in len(x[0]) ]

def set_dispersion(self, parameter, dispersion): …
```

C++ models

Important meta-data used for automatically generating the python wrapper

Defines the parameter for python wrapper generation

cylinder.h

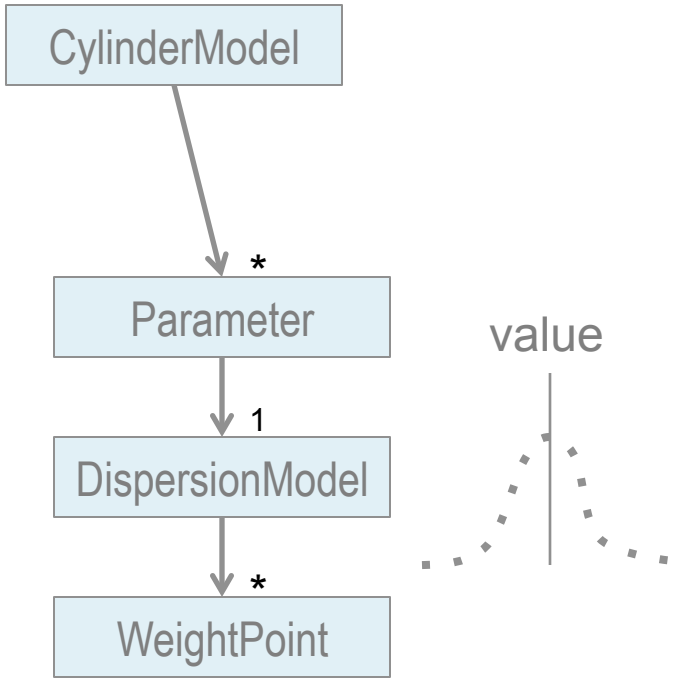
```
#include "parameters.hh"

//[PYTHONCLASS] = CylinderModel
//[DISP_PARAMS] = radius
//[DESCRIPTION] = "this is a model"
//[FIXED] =
//[ORIENTATION_PARAMS] =

class CylinderModel {
public:
  //  [DEFAULT]=radius=20.0 Angstrom
  Parameter radius;

  CylinderModel();

  double operator()(double q);
  double operator()(double qx, double qy);
};
```

cylinder.cpp

```
#include "parameters.hh"
#include "cylinder.h"

CylinderModel::CylinderModel() {
  radius = Parameter(20.0, true);
};

CylinderModel::operator()(double q) {
  return f(q);
};
```

True here means that this parameter can have polydispersity

Parameter class



Available dispersion models:

- GaussianDispersion
- LogNormalDispersion
- RectangleDispersion
- SchultzDispersion
- ArrayDispersion

parameters.hh

```
class Parameter {
public:
  /// Current value of the parameter
  double value;
  /// True if the parameter has a minimum bound
  bool has_min;
  /// True if the parameter has a maximum bound
  bool has_max;
  /// Minimum bound
  double min;
  /// Maximum bound
  double max;
  /// True if the parameter can be dispersed or averaged
  bool has_dispersion;

  /// Pointer to the dispersion model
  DispersionModel* dispersion;

  Parameter();
  Parameter(double);
  Parameter(double, bool);

  /// Method to set a minimum value
  void set_min(double);
  /// Method to set a maximum value
  void set_max(double);
  /// Method to get weight points for this parameter
  void get_weights(vector<WeightPoint>&);
  /// Returns the value of the parameter
  double operator()();
  /// Sets the value of the parameter
  double operator=(double);
};
```

Polydispersity in C++

Remember this?

cylinder.h

```
#include "parameters.hh"

class CylinderModel {
public:
  //  [DEFAULT]=radius=20.0 Angstrom
  Parameter radius;
};
```

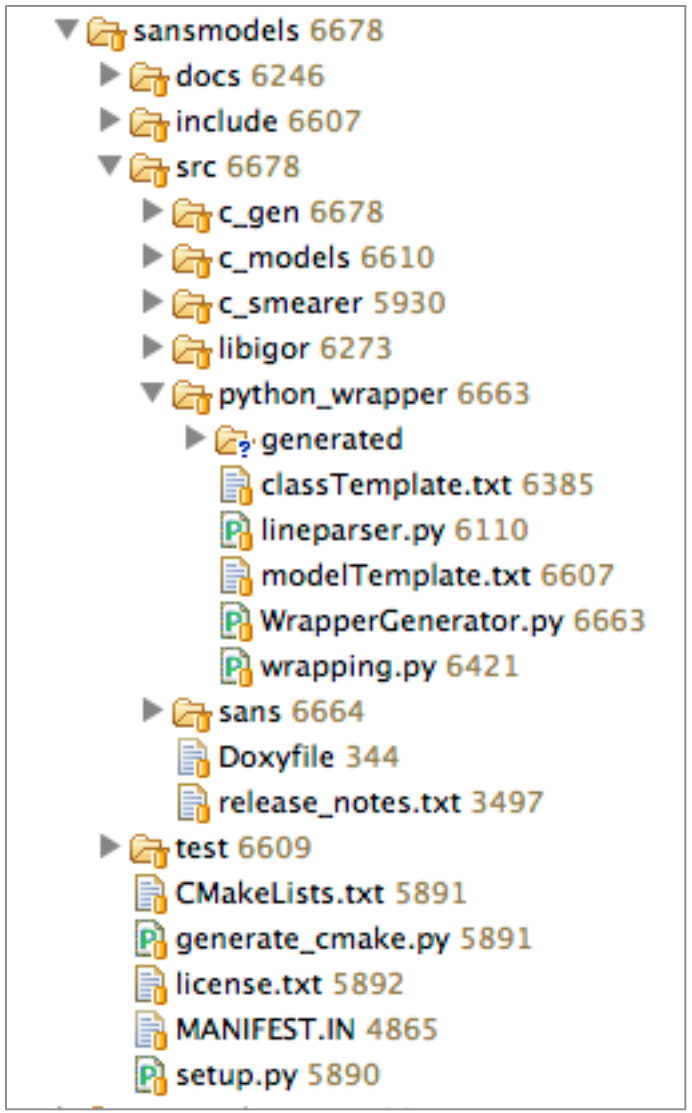
It gives us the weight points to average over

```
double CylinderModel::operator()(double q) {

  // Get the nominal value of the parameter
  double radius_value = radius();

  // Get dispersion points
  vector<WeightPoint> weights;
  radius.get_weights(weights);

  // Loop over radius weight points
  double sum = 0.0;
  double norm = 0.0;
  for(size_t i=0; i<weights.size(); i++) {
    radius_value = weights[i].value;
    sum += weight[i].weight*f(radius_value, q);
    norm += weight[i].weight;
  }
  return sum/norm;
};
```

cylinder.cpp

Exposing C++ models to python

- The setup.py installation takes care of exposing your C++ models to python.
- It generates a C++ wrapper and the python model using templates.
- It pulls most of the information it needs from the header file (cylinder.h)



Exposing C++ models to python

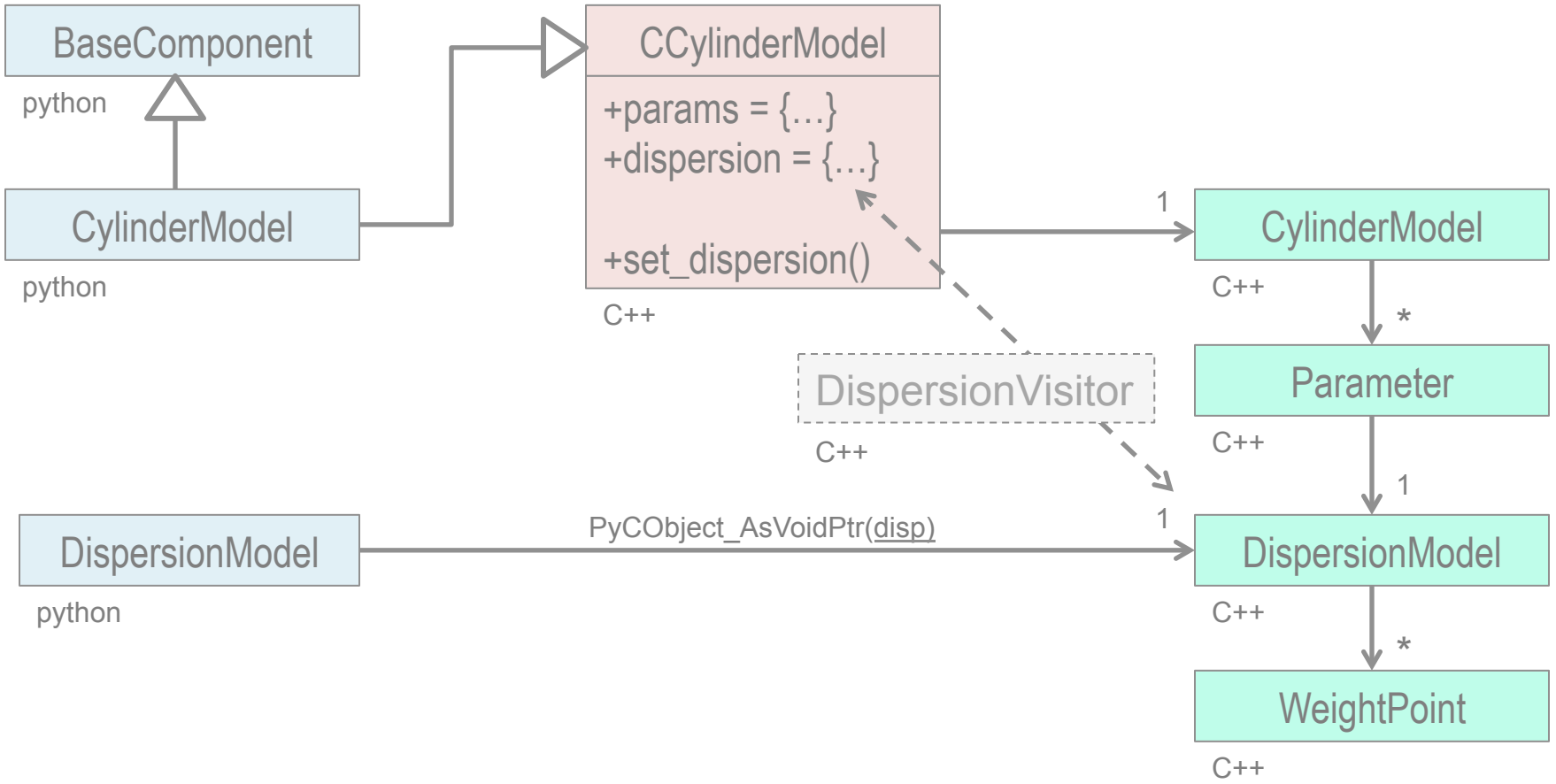
- You don't need to write the python model yourself

```
class CylinderModel(CCylinderModel, BaseComponent):

def __init__(self):
    self.name = 'Cylinder'
```

- Polydispersity is also taken care off

```
from sans.models.CylinderModel import CylinderModel
from sans.models.dispersion_models import GaussianDispersion

m = CylinderModel()
d = GaussiantDispersion()
m.set_dispersion('radius', d)

m.model.dispersion['radius']['width'] = 0.25
m.model.dispersion['radius']['npts'] = 100

m.evalDistribution(numpy.asarray([0.001, 0.002]))
```

Everything you ever wanted to know about C++ models



Development tips

A long time ago Paul Kienzle and I wrote “Team Rules” for new developers. They included things like:

- Golden Rule: committed code should NEVER break
- Silver Rule: never commit to a ‘release’ branch directly
- Follow PEP-8: www.python.org/dev/peps/pep-0008
- Golden Rule of Error Handling: if you catch an exception and all you do is calling pass or print, you have not dealt with the error properly