The *DIALS* Framework

James Parkhurst

Overview

- DIALS aims to provide a framework for developing integration algorithms.
- Users should be able to develop their own algorithms and be given a simple way to enable their algorithm within the framework.
- Development of the framework has so far been incidental to the development of the basic algorithms: a proper design is needed!
- This talk will attempt to describe:
  - The current state of the framework
  - Proposed future developments

The DIALS Framework

CURRENT STATUS

Current Status

- A basic 'framework' is in place with the following components:
  - *dxtbx* and experimental models
  - Crystal model
  - Reflection 'bucket' container
  - Factory methods to create major components
  - Script class to manage common command line application configuration
  - Algorithm configuration via *phil* parameters

dxtbx and experimental models

- The *dxtbx* experimental models are used throughout to provide data access.
- The ImageSet/ImageSweep classes are used to provide access to image data and experimental models.
- Some algorithms are configured to alter their behaviour based on whether they are given a sweep (rotation data) or an imageset (sequence of still images).*

* Only the spot finder does this at the moment!

Factory functions

- Top-level algorithms (spot finder, refinement and integration) are configured via factory functions.
- The input parameters are given in the form of *phil* parameters.
- This allows simple instantiation of the requested algorithm from the input parameters as shown in the example.
- Current implementation is not easily extendable: new algorithms need to be added to a large "if" statement.

Factory functions

```
class IntegratorFactory(object):

    @staticmethod
    def from_parameters(params):
        # Configure the algorithms to extract reflections, compute the
        # background intensity and integrate the reflection intensity
        compute_spots = IntegratorFactory.configure_extractor(params)
        compute_background = IntegratorFactory.configure_background(params)
        compute_centroid = IntegratorFactory.configure_centroid(params)
        compute_intensity = IntegratorFactory.configure_intensity(params)
        correct_intensity = IntegratorFactory.configure_correction(params)

        # Return the integrator with the given strategies
        return Integrator(compute_spots = compute_spots,
                          compute_background = compute_background,
                          compute_centroid = compute_centroid,
                          compute_intensity = compute_intensity,
                          correct_intensity = correct_intensity)

# Instantiate the integrator
integrator = IntegratorFactory.from_parameters(params)
```

The IntegratorFactory class

Command line script classes

- Enables simple consistent command-line script creation.
- ScriptRunner class sets up default command line options and logging, and loads global *phil* parameters.
- Scripts inherit from the ScriptRunner class and overload the "main" function to perform their processing.

*There are (non standard library) python packages to do this (pycli, cement). Could be worth looking at, although would introduce another dependancy.

Command line script classes

```
class Script(ScriptRunner):

    def __init__(self):
        usage  = "usage: %prog [options] [param.phil] " \
                 "sweep.json crystal.json [reference.pickle]"

        ScriptRunner.__init__(self, usage=usage)

        # Add a command line parameter
        self.config().add_option(
            '-o', '--output-filename',
            dest = 'output_filename',
            type = 'string', default = 'integrated.pickle',
            help = 'Set the filename for integrated reflections.')

    def main(self, params, options, args):

        # Print the command line help
        if len(args) != 2:
            self.config().print_help()
            return

        # Do the processing
        integrate = IntegratorFactory.from_parameters(params)
        sweep = load.sweep(args[0])
        crystal = load.crystal(args[1])
        reflections = integrate(sweep, crystal)
```

Example using the ScriptRunner class

Configuration via *phil* parameters

- The framework is configured using *phil* parameters.
- Uses master files located in dials/data to provide default configuration.
- Also looks, by default, for a ~/.dialsrc file where users can override the default parameters.
- Rudimentary bash completion for phil parameters can be enabled by running the following command:

```
. `libtbx.show_build_path`/dials/completion
```

Issues that need addressing

- Framework is currently not user-extensible
  - Requires modification of factory methods to add new algorithms and modification of global configuration files to add new parameters.
- Reflection container has become bloated
  - Too much different data being stored in single class (i.e. not specialized enough).
- Current detector model does not represent hierarchical detectors well
  - Multi-panel detectors are represented as an array of panels, thereby losing information about the experimental setup.

The DIALS Framework

PROPOSED DEVELOPMENTS

Hierarchical detector model

1D Representation

Implemented in C++. Used in the majority of cases. Panels are accessed like a standard array.

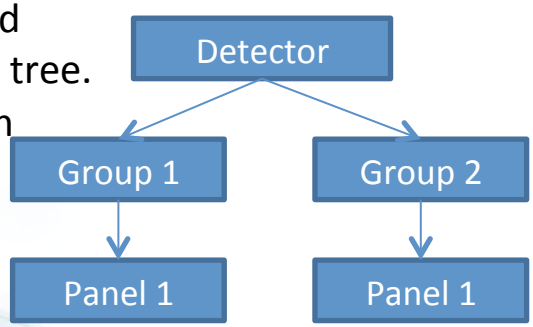

```
# Add panels to the detector
detector = Detector()
panel1 = detector.add_panel()
panel1.set_name("Panel 1")
panel1.set_type("Panel")

panel2 = detector.add_panel()
panel2.set_name("Panel 2")
panel2.set_type("Panel")

# Access a panel
p = detector[0]
```

Hierarchical Representation

Implemented in python. Accessed explicitly through a method of the detector. Allows creation of general hierarchy of groups and panels in the form of a tree. Mainly used in creation and refinement. Panels must be added explicitly to hierarchy.




```
# Create the hierarchy
root = detector.hierarchy()
root.set_name("Detector")
root.set_type("Detector")

group1 = root.add_group()
group1.set_name("Group 1")
group1.set_type("Group")
group1.add_panel(panel1)

group2 = root.add_group()
group2.set_name("Group 2")
group2.set_type("Group")
group2.add_panel(panel2)
```

More specific data models

- Current reflection container is messy and bloated.
- Container is often saved with some fields empty and could lead to confusion (e.g. all reflections from the spot prediction routine are saved with miller index (0, 0, 0)).
- Could split contents into more targeted data models that provide methods to act on their specific data as shown in the table below.

Model	Description
Prediction	Miller indices, rotation angles and detector coordinates etc
Observation	Centroids and intensities.
Shoebox	Bounding box, pixel values, mask and background.

<u>Graeme is currently designing a better, templated, reflection container.</u>

An extensible plugin framework

- Aim to replace existing factory functions with an extensible plugin framework
- Features:
  - High level interface to major components (e.g. Spot finding, indexing, refinement and integration)
  - Low-level interfaces to provide specific functionality
  - Automatic registration of extensions
  - Automatic discovery of extensions
  - Automatic extraction of configuration parameters

High-level interfaces

- High-level components (e.g. spot finding, indexing, refinement and integration) are implemented according to an iterative scheme (taken from xia2).
- Allows program to make decisions to re-process data based on the output of its data processing.
- Logic is handles in the toplevel.Interface class with abstract methods. Instances override these methods as shown below.

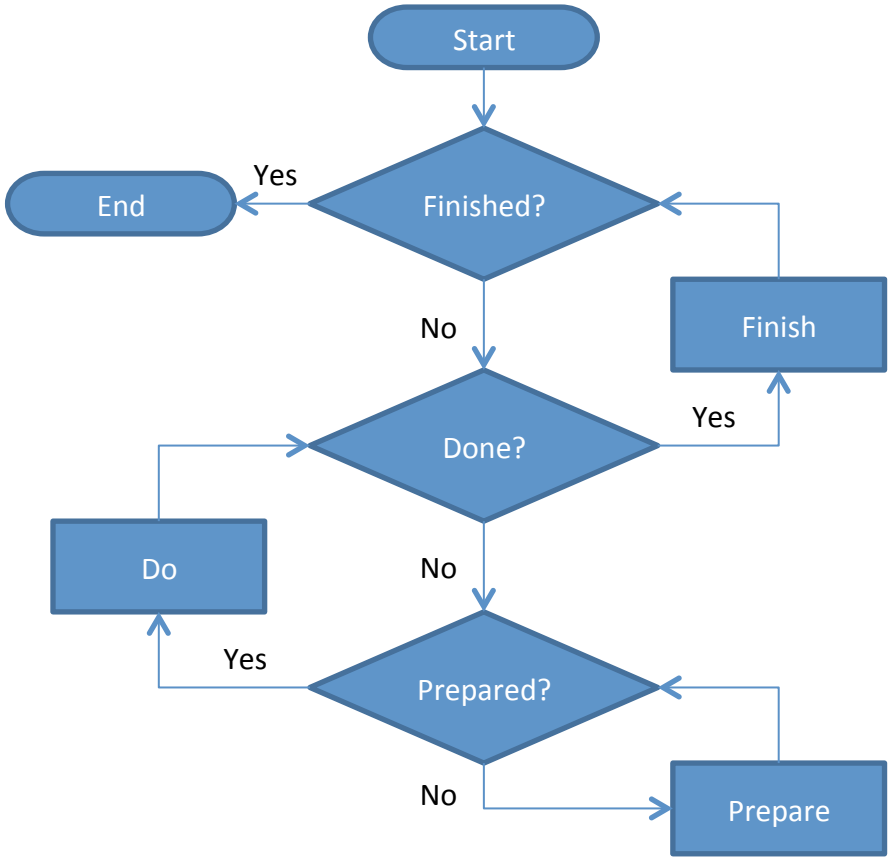
```
class Integrator(toplevel.Interface):

    def prepare(self):
        self.extract_shoeboxes()
        self.prepared = True

    def process(self):
        self.compute_background()
        self.compute_centroids()
        self.compute_intensities()
        self.processed = True

    def finish(self):
        self.finished = True
```

Simple high-level integrator instance example

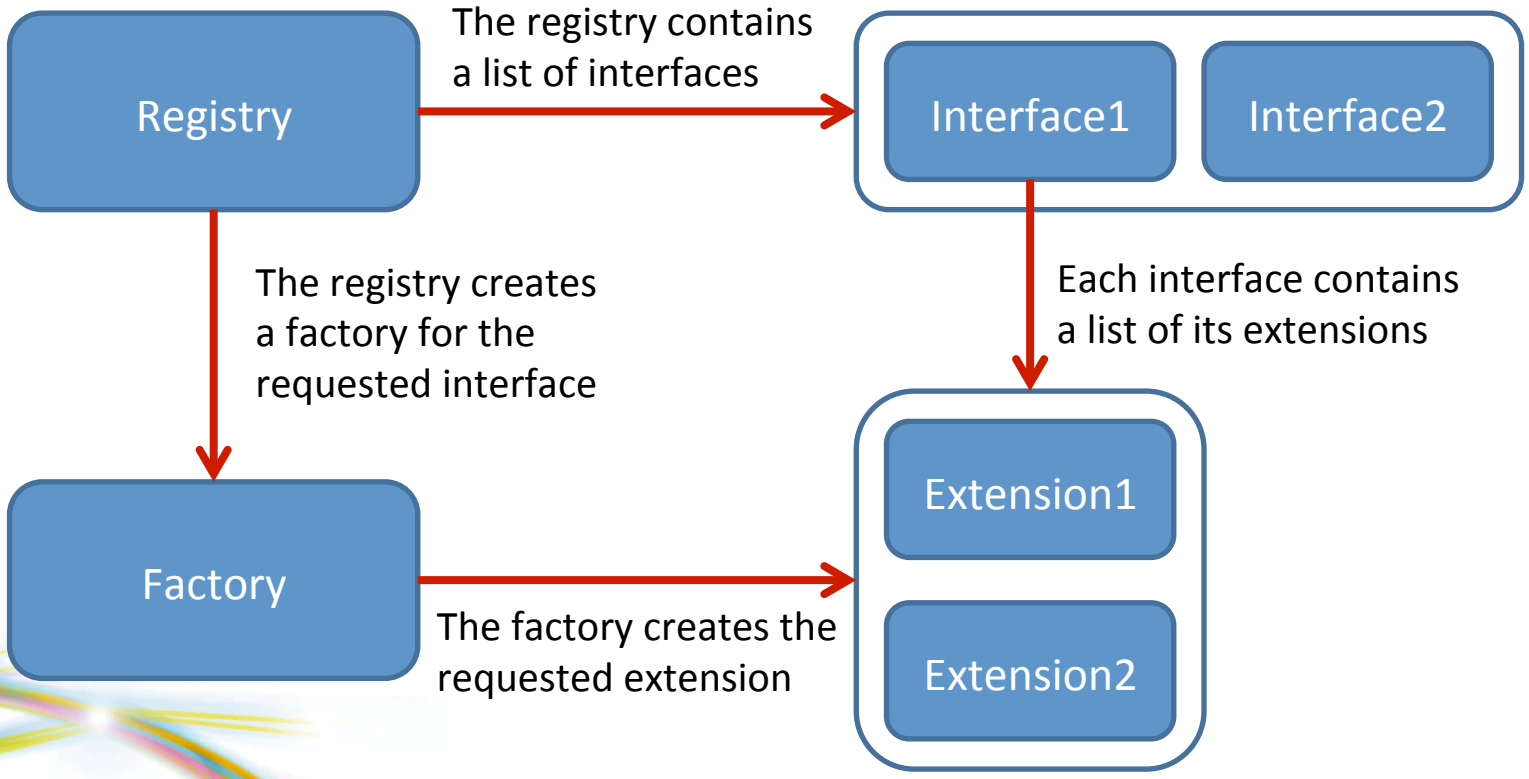


Flow chart of high-level interface (taken from xia2)

Low-level plugin interfaces

- Aim is to allow algorithms to be written by users that add specific functionality without modifying the framework code.
- Interfaces for algorithms are specified using the "Interface" meta class. Abstract methods are specified to ensure extensions implement required functionality.
- Extensions are created by inheriting from the interface they extend.
- Extensions are instantiated using a factory function obtained via a global registry of interfaces and extensions.
- The extension's meta data (including *phil* parameters) are defined in the class itself and automatically extracted allowing master *phil* specs to be generated on the fly.

Data model


Registry
The registry contains
a list of interfaces
Interface1
Interface2
The registry creates
a factory for the
requested interface
Each interface contains
a list of its extensions
Factory
Extension1
Extension2
The factory creates the
requested extension

Example

Creating an interface

Interfaces are declared with the “Interface” metaclass and inherit from “object” to gain new-style python class features.

Abstract methods must be overridden by the extension.

```
class Integrator(object):
    '''Description of interface'''
    __metaclass__ = Interface
    name = "integrator"

    @abstractmethod
    def integrate(self):
        pass
```

Creating an extension

Extensions inherit from their interface.

A name, *phil* parameters and other meta data are given to aid configuration.

Abstract methods must be are overridden.

```
class MyIntegrator(Integrator):
    '''A description of the extension'''

    name = "my_integrator"

    phil = '''
        param1 = 0
          .type = int
          .help = "a parameter"
    '''

    def __init__(self, sweep, crystal, params):
        super(MyIntegrator, self).__init__()

    def integrate(self):
        pass
```

Instantiating an extension

A factory class is created for the interface.

The extension is instantiated by name from the factory.

```
factory = Registry.factory(Integrator)

integrator = factory.create("my_integrator",
    sweep, crystal, params)
integrator.integrate()
```

Summary

- Current status
  - Factory methods to configure algorithms with user selected methods.
  - Phil parameters used to configure algorithm
  - Helper classes to write command line scripts
- Proposed future developments
  - New hierarchical detector model
  - Improved data models
  - An extensible plugin framework

Interface Meta Class and Factory Class

Interface Class

```
class Interface(ABCMeta):
    def __init__(self, name, bases, attrs):
        super(Interface, self).__init__(name, bases, attrs)

        if 'name' not in self.__dict__:
            raise RuntimeError("%s has no member 'name'" % name)

        if not hasattr(self, '__registered__'):
            self.__registered__ = True
            Registry.add(self)
```

Factory Class

```
class Factory(object):

    def __init__(self, plugins):
        self._plugins = plugins

    def create(self, name, *args, **kwargs):
        return self._plugins[name](*args, **kwargs)
```

Registry Class

```
@singleton
class Registry:
    def __init__(self):
        self._interfaces = set()
    def add(self, iface):
        self._interfaces.add(iface)
    def clear(self):
        self._interfaces.clear()
    def remove(self, iface):
        self._interfaces.remove(iface)
    def __len__(self):
        return len(self._interfaces)
    def __iter__(self):
        return iter(self._interfaces)
    def __contains__(self, iface):
        return iface in self._interfaces
    def extensions(self, cls):
        if cls not in self:
            raise TypeError('interface %s is not registered' % cls)
        stack = list(cls.__subclasses__())
        while len(stack) > 0:
            cls = stack.pop()
            yield cls
            stack.extend(cls.__subclasses__())
    def all_extensions(self):
        return dict((iface, list(self.extensions(iface))) for iface in self)
    def factory(self, iface):
        return Factory(dict((sc.name, sc) for sc in self.extensions(iface)))
```

Top-level interface class

```
class Interface(object):

    __metaclass__ = ABCMeta

    def __init__(self, maxiter=20):
        self._prepared = False
        self._processed = False
        self._finished = False
        self._maxiter = maxiter

    def run(self):
        count1 = 0
        while not self.finished:
            count2 = 0
            while not self.processed:
                count3 = 0
                while not self.prepared:
                    self.prepare()
                    count3 += 1
                    if count3 >= self._maxiter:
                        raise RuntimeError('maximum prepare iterations reached')
                self.process()
                count2 += 1
                if count2 >= self._maxiter:
                    raise RuntimeError('maximum process iterations reached')
            self.finish()
            count1 += 1
            if count1 >= self._maxiter:
                raise RuntimeError('maximum finish iterations reached')
```