Measurement Framework Overhaul Release Notes

In the next major release of the LSST stack, the system for measuring the properties of sources will be replaced by a new one, housed mostly in the new `meas_base` package. In addition to providing new `Task` classes and a new plugin interface for measurement algorithms, this overhaul also includes changes to the schemas of the catalogs produced by the measurement framework

New Tasks

The main entry point for the new measurement framework is the new `SingleFrameMeasurementTask` class (in `meas_base`), which is intended as an *almost* drop-in replacement for the current `SourceMeasurementTask` (in meas_algorithms). It's not drop-in replacement, because it has different plugins, a slightly different slot system, and an entirely different output schema - so we will use it everywhere SourceMeasurementTask was used before, but when changing other configuration settings from their defaults - particularly those that deal with the measurement algorithms or their outputs, you'll need to be careful to use configuration values that are consistent with the measurement tasks you're useing. We've provided configuration instructions to make that easy, as described in the next section.

`SingleFrameMeasurementTask` combines the work previously done by two classes, the old `SourceMeasurementTask` and the C++ `MeasureSources` class. It initializes the plugins (which determines the schema) in its constructor, then invokes the plugins on each source in the image, replacing neighbors with noise as it does so (this is delegated to the `NoiseReplacer` class, which replaces the `ReplaceWithNoiseTask` subtask). Unlike `SourceMeasurementTask`, `SingleFrameMeasurementTask` does not do forced photometry (see below). Sources are also processed in a slightly different order (see Simultaneous Multi-Object Measurement).

Schema Changes and Versioning

The changes to `afw::table`'s `Schema` objects to support the new measurement framework are handled by a new versioning mechanism, which allows the old behavior (necessary for the old framework) to coexist with the new behavior in the stack. A `Schema` with version=0 corresponds to one appropriate for use with the old `SourceMeasurementTask`, while version>0 corresponds to `SingleFrameMeasurementTask` and the new forced photometry tasks. Here are the differences between versions:

- In version 0, periods in field names were translated to underscores when writing to FITS, making it impossible to use underscores in field names. This was due to a misunderstanding of the FITS standard, and was never necessary. This translation is not done for version>0 schemas, and hence both periods and underscores will work with the code.
- In version>0, our naming conventions use underscores instead of the periods used in version 0. So while both are permitted by the code in version>0 (which allows us to read external FITS files with fewer restrictions, and note that periods will not be treated as a delimiter in version>0), only underscores should be used in LSST code, and underscores will now be used as the delimiter by the `SubSchema` class and `Schema::operator[]` operator when joining pieces of field names.

In version>0, we have stronger naming convention for fields generating by plugin algorithms, which tie the plugin name and location to its outputs. These conventions are as follows:

- Names should start with "<package-abbr>_<class>", where "<package-abbr>" is an abbreviated version of the package name that provides enough information to specify the package location. For instance, plugins in `meas_base` use "base" as their prefix, while those in `ip_diffim` would use simply "diffim". The "class" part of the name is simply the name of the class that defines the algorithm, minus any explict "Plugin" or "Algorithm" suffix. For instance, the field name prefix used by `meas_base`'s `SdssShapeAlgorithm` class would be "base_SdssShape".
- Flux measurements should end with "_flux", even if that's repetitive with the name of the class (e.g. "base_PsfFlux_flux").
- Position measurements should consist of two fields, ending in "_x" and "_y".
- Ellipse measurements should be saved using the "Quadrupole" parametrization (even if they aren't measured as moments), using fields ending with "_xx", "_yy", and "_xy".
- Uncertainties on individual fields should be saved with names that end in "_<p>Sigma", where "<p>" is the suffix of the field the uncertainty corresponds to (e.g. "_fluxSigma" or "_xSigma").
- Uncertainty covariances should be saved with names that end in "_<p1>_<p2>_Cov", such as "_x_y_Cov" for the uncertainty on "_x" and "_y".
- Flag fields should have names like "_flag_<reason>", where "reason" is a camelCase string identifying what went wrong. A special flag that is simply the algorithm prefix + "_flag" is additionally set for any fatal error (but this will likely be changed on

  ⚠ DM-464 - Jira project doesn't exist or you don't have permission to view it.

  prior to the S14 release).

Aliases and Slots

We have added an alias feature to the `Schema` class, via an `AliasMap` object it holds. Aliases are handled simply as an extra stage in `Key` lookup - when a `Key` is requested via a string, its beginning is compared against all aliases to see if it should be replaced. Partial replacements are allowed, but only at the beginning, and multiple replacements are not. For example, if the `AliasMap` includes the mapping "slot_PsfFlux"->"base_PsfFlux", then a `Key` lookup on "slot_PsfFlux_flux" will resolve to "base_PsfFlux_flux", but a lookup on "r_slot_PsfFlux_flux" will not be affected. Because `Key` lookup happens implicitly when getting a value from a record via string, this makes it even more important to do `Key` lookup once in advance rather than use a string to access the same value repeatedly.

The alias system is available for both version 0 and version > 0 tables, and is now used to define the slot mechanism in SourceTable/SourceRecord /SourceCatalog in both cases; each slot corresponds to a predefined field name prefix that is mapped to an actual measurement via an alias. These predefined names are the name of the slot, starting with a capital letter (e.g. "PsfFlux", "Centroid"), prefixed with "slot_" (or "slot." for version 0). For instance, a typical AliasMap containing slot definitions would have the following mappings for version 1:

- slot_PsfFlux -> base_PsfFlux
- slot_ApFlux -> base_SincFlux
- slot_Centroid -> base_SdssCentroid
- slot_Shape -> base_SdssShape

Slot definitions are now persisted simply as aliases, with slots saved via older versions of the stack translated into aliases when read from disk. Changing one of these aliases in an `AliasMap` attached to a `SourceTable` will notify the table that slots have changed and the cached Keys that correspond to them must be updated (in fact, the old slot definers, such as `SourceTable::definePsfFlux()` are now implemented by simply changing the alias and letting the notification callback do its work).

Note that this means that slot values can now be accessed via string, just like any normal measurement.

Forced Photometry

The new measurement framework adds several new `Task` classes for forced photometry, including the capability to perform forced photometry on coadds (long present on the HSC fork of the codebase but absent on the LSST side). The new Tasks are:

- `ForcedMeasurementTask`, an analogue of `SingleFrameMeasurementTask` for forced measurements, used as a subtask by the command-line tasks below. Users should rarely if ever have to use it directly, aside from configuring it as a part of those command-line tasks.
- `BaseReferencesTask` and `CoaddSrcReferencesTask`, which (respectively) define an interface for retrieving reference objects that correspond to a patch of sky and implement that interface using `ProcessCoaddTask` outputs as the reference objects.
- `ForcedPhotImageTask`, `ForcedPhotCcdTask`, and `ForcedPhotCoaddTask` are command-line driver tasks that mirror pipe_task's ProcessImageTask (base class; most of the implementation), ProcessCcdTask (specialization for CCD-level processing), and `ProcessCoaddTas` (specialization for coadd-level processing). These delegate most of their work to the above two subtasks.

More information on forced photometry can be found in the Doxygen documentation for the `meas_base` package.

Simultaneous Multi-Object Measurement

While the old `SourceMeasurementTask` processed sources in the order set by the catalog it was given, `SingleFrameMeasurementTask` and `ForcedMeasurementTask` instead iterate over deblend families; within each family, the children are processed individually first, followed by the parent. After the individual source measurements, plugins are then given an opportunity to measure all the children at once at once, using an interface that was not present in the old framework, using the same pixel values used in the parent measurement but writing outputs to per-child records. This allows us to support plugins that fit multiple objects simultaneously (possibly using the earlier non-simultaneous measurements as input).

This will likely provide the easiest route to properly-deblended forced photometry; rather than attempt to transform deblender outputs from frame to frame, or rely on a full, multi-epoch deblender, we can simply fit families simultaneously in forced mode. This does not, of course, address the problem of consistent multi-band detection or deblending.

Currently we have no plugins that use this API. Of our current algorithms, only the PSF flux and the galaxy model flux are likely to ever be usable in this mode.

Python and C++ Measurement Plugins

While the new framework allows plugins to be written in Python, we expect that essentially all production plugins will be implemented in C++. As a result, we've put much more effort into reducing the amount of boilerplate necessary to implement a C++ plugin (see the `meas_base` Doxygen documentation) than we have into reducing boilerplate for pure-Python plugins. Even so, we're not entirely happy with this C++ interface, and plan to investigate some alternate designs in W15. We also plan (in W15) to give the pure-Python plugin-writing experience much more attention - while the Python plugin interface itself will not change, we plan to add helper classes that will make it easier to implement new plugins.

Error-Handling and Diagnostics

One of the main areas we hope to improve on in the new framework is in the handling of errors in measurement algorithms. Our first goal in this area is to ensure that no errors go unreported in detail: known failure modes should be reported as problem-specific flags, and any unexpected exception should result in a warning-level log message that would enable the problem to be tracked down as a bug. These known failures are indicated by throwing an instance of `lsst.meas.base.MeasurementError`, which stores information about a problem-specific flag that should be set. All other exceptions that propagate up to the measurement task will be logged as warnings. For configuration errors that affect all sources to be measured on an exposure (i.e. running an algorithm that requires a `Psf` on an `Exposure` with no `Psf`), we will also provide custom exceptions that will be treated as fatal by the measurement framework (see

> ⚠ DM-461 - Jira project doesn't exist or you don't have permission to view it.

).

For measurements accessed via slots, however, problem-specific flags are not available, and the user is forced to rely only on the "general failure" flag that is common to all algorithms. This flag is often ambiguous, however, as it is used both when an algorithm has completely failed and has no valid output, and when a more minor error occurred that yields a valid but slightly less trustworthy result (for example, the SdssShape algorithm typically uses Gaussian-weighted moments, but can fall back to unweighted moments). To address this, we plan to add a "general suspect" flag to all algorithms and the slot system, in addition to the "general failure" flag. The failure flag (likely renamed to "*_flag_failed" from simply "*_flag") will be used to indicate a complete failure, and the new "suspect" flag ("*_flag_suspect"). This is slated for

DM-464 - Jira project doesn't exist or you don't have permission to view it.

.

Missing Features and Known Problems

For issues that we expect to complete in S15, please consult

DM-1769 - Jira project doesn't exist or you don't have permission to view it.

and

DM-85 - Jira project doesn't exist or you don't have permission to view it.

. These include reimplementing aperture corrections, removing now-deprecated afw::table features, and reimplemented database ingest.