Data Units and Graph-Based Inputs to defineQuanta

This page is a design proposal to address the problems in the baseline `SuperTask` noted in the last section of SuperTask Status: Science Pipelines Perspective. In this design, we replace the camera-dependent dictionary data IDs currently used by the Butler with a more structured set of classes that are camera-independent. Camera-specific data ID keys will still be relevant in the user-provided expressions that describe that output datasets should be produced and what input datasets should be considered, but these expressions will be explicitly transformed into to a new data structure containing only camera-independent classes before they come into contact with any `SuperTask`.

Units and Datasets

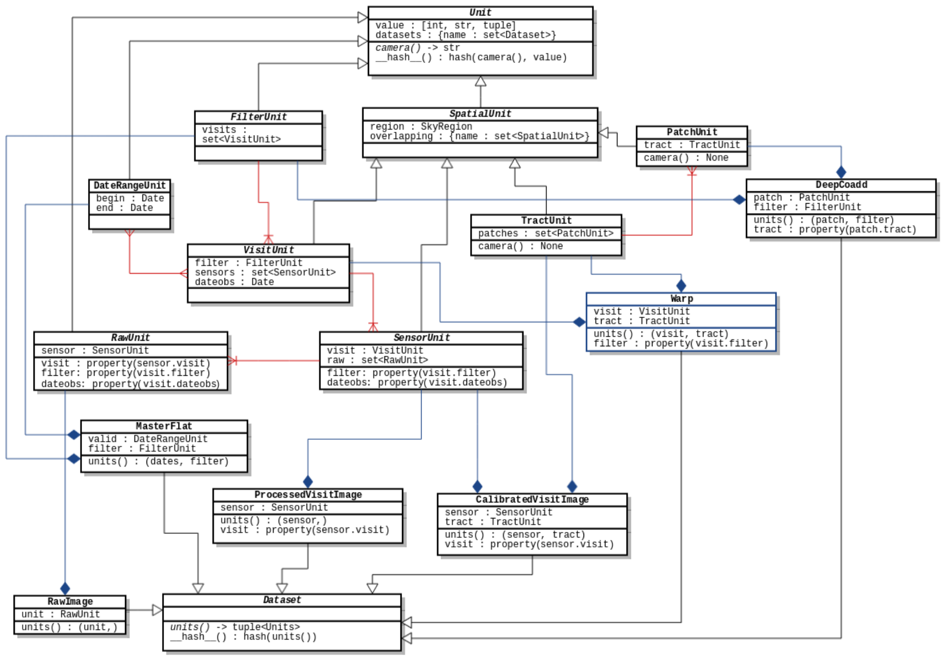
A *`Unit`* is a conceptually similar to a data ID key (i.e. "visit", "ccd", "patch"), and a particular unit may (for some cameras) map exactly to an existing data ID key. Units, however, are considered to be generically important quantities from an algorithmic perspective, and hence they are camera-generic. A Dataset is analogous to a combination of a `datasetType` and its data ID; a Dataset class corresponds to a `datasetType`, while an instance adds a tuple of `Unit` instances that represent the data ID in a camera-independent way.

In code, `Unit` is an abstract base class that holds an immutable unique identifier whose type is defined by the subclass, such as an integer ID for visits, a tuple of integers for patches, or a string name for filters. The actual types and values of these IDs are unimportant, and may in fact be camera-dependent, as long as the values are globally unique identifiers within a camera. A `Unit` also holds a dictionary of all the datasets that are defined on that unit (i.e. all of the `ProcessedVisitImages` for a particular visit), and has an abstract `camera()` method that must be implemented by concrete derived classes. `Un it` subclasses define attributes that link one Unit to its related Unit instances (e.g. the visit that a sensor belongs to, and vice versa).

The set of **all** `Units` that may be used to define a `Dataset` is shown in the the diagram below. Some of these classes (with italics) are themselves abstract, and need to be subclassed in an obs package to create a concrete Unit class. Only the `TractUnit` and `PatchUnit` classes are used directly by all cameras. `Datasets` and `SuperTasks` may interact only with the classes defined here, however; per-camera specializations will be used only to make globally-unique identifiers and to map `Units` to the per-camera keys we use in data IDs today. It is, of course, quite likely that I've actually missed one or two important kinds of `Unit`, but it's highly unlikely that number is > 5, and the important point is that the set of all `Unit` types is itself an important part of this abstraction.

Some `Units` inherit from the intermediate base class `SpatialUnit`, which represents a Unit that occupies a specific area on the sky. It holds a conservative (inclusive) approximation of that region as an attribute (probably something like a `sphgeom.RangeSet` in a particular pixelization), and a dicti onary linking it to all of the `SpatialUnits` whose regions overlap its own. Like the other link attributes, this dict may be empty or have some possible relationships missing.

The diagram below only attempts to show a few examples of concrete `Datasets`. Unlike the set of all `Unit` types, the set of all `Dataset` types is very much dynamic: some `Datasets` may be defined only when a `SuperTask` is configured; the input and output datasets of a `SuperTask` will typically be defined by a special `pex_config` field that has a user-configurable name and a SuperTask-defined set of units. It may even be better to not define a separate derived type for every `Dataset`, and instead have a single flexible data structure with e.g. `__getattr__` overloading for attributes that allow it to be used for any `Dataset`; I consider that an implementation detail. I think it's easier for the conceptual discussion to think of them as classes.



DatasetGraph and the new SuperTask signatures

Because of the links between them, a set of related `Unit` and `Dataset` instances naturally forms a graph-like structure. We define the `DatasetGraph` class as a top-level entry point to that graph:

```
class DatasetGraph:



        def __init__(self):
        self.datasets = {}  # dict of {name: set<Dataset>}
        self.units = {}     # dict of {name: set<Unit>}
```

A `DatasetGraph` could be used to represent all of the content in a `DataRepository` (including its parent repositories), but it can also be bounded in two ways:

- It may not contain all possible `Units` (which would require some link attributes on included Units to be None).
- It may not contain all possible `Datasets` (which just makes some set attributes smaller).

However, a well-formed `DatasetGraph` may not have any *`Datasets`* with `Unit` attributes set to `None`.

A `DatasetGraph` can also be used to represent a superset of the contents of a `DataRepository`: new datasets can be added to represent outputs that have not yet been generated.

As a result, the right `DatasetGraph` represents everything `defineQuanta` needs: given a `DatasetGraph` that is bounded to the intersection of the input and output data ID expressions, `defineQuanta` can iterate over its input datasets and add instances of its output dataset to the graph, using the link attributes to form groups between related datasets and build Quanta. This also works for a collection of SuperTasks in a `SuperTaskComposite`: we can iterate forward (i.e. the same direction as processing flow) through the `SuperTasks`, adding new output datasets. Because the `DatasetGraph` is bounded by both the input and output data ID expressions, no Quanta for unnecessary outputs will be generated. We thus propose the following signatures for `SuperTask`'s core methods:

```
class Quantum:


    def __init__(self, inputs, outputs):
        """Construct from dicts of input and output datasets.


        Parameters
        ----------
        inputs : dict of {name: set<Dataset>}
            All inputs used by a single invocation of a SuperTask.
        outputs : dict of {name: set<Dataset>}
            All outputs produced by a single invocation of a SuperTask.
            """
        self.inputs = inputs
        self.outputs = outputs




class SuperTask:

    def defineQuanta(self, datasets):
        """Return a list of Quantum objects representing the inputs and outputs
        of a single invocation of runQuantum().

        Parameters
        ----------
        datasets : DatasetGraph
            A DatasetGraph containing only Units matching the data ID
            expression supplied by the user and Datasets that should be
            present in the repository when all previous SuperTasks in the same
            pipeline have been run.  Any Datasets produced by this SuperTask
            should be added to the graph on return.

        """
        raise NotImplentedError()

    def runQuantum(self, quantum, butler):
        """Run the SuperTask on the inputs and outputs defined by the given
        Quantum, retrieving inputs and writing outputs using a Butler.
        """
        raise NotImplentedError()

    def getDatasetClasses(self):
        """Return a dict containing all of the concrete Dataset classes used
        by this SuperTask.
        """
        result = {}
        for fieldName in self.config:
            cls = getattr(self.ConfigClass, fieldName)
            if subclass(cls, DatasetField):
                p = getattr(self.config, fieldName)
                result[p.name] = p.type
        return result

    def getUnitClasses(self):
        """Return a dict containing all of the concrete Unit classes used
        by this SuperTask.
        """
        result = {}
        for DatasetClass in self.getDatasetClasses().values():
            for UnitClass in DatasetClass.UnitClasses:
                result[UnitClass.name] = UnitClass
        return result
```

The `getDatasetClasses` and `getUnitClasses` methods provide introspection into the `Dataset` and `Unit` types used by the `SuperTask`. The implementations above assume a `DatasetField` (inherits from `pex.config.Field`) is used in the `SuperTask`'s config class to define each of its input and output datasets. A sketch of `DatasetField` itself is attached (but it may require a familiarity with `pex.config` and Python metaprogramming with descriptors to understand).

These interfaces have a few more advantages over the baseline interface in terms of what they demand from Butler:

- We do not require a `Butler` at all in `defineQuanta`, which means it is not strictly necessary for each activator to be able to construct a "pre-flight Butler" at all when the pre-flight environment is significantly different from the job execution environment.
- The only methods we need on a job execution `Butler` are `get(dataset)` and `put(value, dataset)`.

Given a `SuperTaskComposite`, a `PreFlightActivator` will be able to inspect the constituent `SuperTasks` to build a list of all `Dataset` types used by the composite, and from that, it can build a list of all `Units` used by the composite. This provides one kind of bound on the `DatasetGraph`: the set of `Unit` types and and `Dataset` types that will be included (and by extension, which `Unit` link attributes will be permitted to be `None`). The much harder bound is the one values of the actual Unit instances present, which must be formed by intersecting the input and output data ID expressions. That was in many respects the hardest problem in the old design, and it remains the hardest problem in this one; all we have accomplished thus far is redefining it as a handshake between the `Unit` hierarchy and the `Butler`, rather than a handshake between the `SuperTaskComposite` and the `Butler`. But this is not an insignificant improvement, because the `Unit` hierarchy contains ~8 explicit relevant classes with well-defined relationships and the set of possible `SuperTasks` is (like the set of `Datasets`) explicitly dynamic.