Data Butler Multiple Repository Design Proposal

Abstract

Butler needs to be able to read from several Repositories and to be able to write to many Repositories.

We also need to plan for git-style branching.

Related Existing Issues

1. Currently we have the concept of a Repository, which we take to mean a collection of datasets with associated Policy (e.g. read/write rules). However there is no formal representation of a Repository in Butler. Instead data on disk is accessed directly by Butler (and CameraMapper). As a result the Repository access logic is somewhat scattered and it is hard to allow for more than one Repository.
2. Data is accessed, that is: read directly, by Butler, Mapper, and Mapping which makes it hard-to-impossible to allow for multiple storage types (posix, database, etc). We need a pluggable way to decouple the storage infrastructure.
3. Object serialization is hard coded in Butler (and framework classes); file formats are hard coded. We need a pluggable way to serialize objects to files of specified formats (FITS, HDF5).

Definitions

Repository

General Overview

Conceptually a Repository is any place where related data is stored. The Repository class implements an interface that is independent of data location (database, posix, in-memory, etc) and format (FITS, HDF5, etc) and manages its relationship to other repositories. Some Repositories are read-write and some are read-only.

Parent

A parent Repository is conceptually part of a Repository (implementation is probably going to be a reference to a separate read-only repository). The parent contains data that is considered to be part of the Repository, but may be masked by data in the Repository. The Parent is read-only.

If a Repository has a parent, the Repository must keep a handle to its parent. At runtime this can be a reference. At load time the implementation depends on the type of Storage.

Input Repository

Input Repositories are read only. Repositories that are loaded from persistence must be Input Repositories. If they are to be changed or added to then a new Repository should be created with the loaded Repository as a Parent.

Output Repository

Output Repositories may be written to (and read from, I think(?)).

Dataset & Aggregate Repository

Repositories may access datasets directly, or they may aggregate other repositories.

The current thinking is that this will be an either-or implementation; a repository will either have exactly one dataset Storage, or it will have references to other repositories.

Repository Classes

The exact class structure is somewhat TBD, and determination will rely on the steps outlined in step 1 of the Refactoring section, below. But, the current idea follows:

DatasetRepository

A kind of Repository that contains access to datasets. DatasetRepositories may be Input or Output Repositories. It can have 0 or 1 parents, and no children.

OutputAggregateRepository

For output purposes an Aggregate Repository writes to other repositories but does not contain any datasets itself; it has child repositories. Children may be any kind of repository (Dataset or Aggregate). By default writes go to all children, but writes may be written to fewer repositories by a method TBD (ideas include dataId, datasetType, policy). It can have 0 or 1 parents.

InputAggregateRepository

If a repository is to have many parents then its parent would be an InputAggregateRepository. Butler mapping searches all the contained parent repositories and will follow normal behaviors (e.g. current behavior is to raise an exception if a single match can't be found for a search). Persisted OutputAggregateRepositories reload as InputAggregateRepositories.

Storage

Storage is a protocol (or abstract base class TBD) that defines the api for concrete Storage classes that implement read and write access. Storage classes can be added by client code and are to be pluggable; i.e. provided by client code.

Concrete classes include support for one of:

- file system (FilesystemStorage or PosixStorage)
- database (DatabaseStorage)
- in-memory (InMemoryStorage)
- stream (StreamStorage)
- others, can be implemented by 3rd party users

Concrete Storage classes are responsible for implementing:

- Concurrency control that cooperates with their actual storage.
- Handle-to-stored-Parent for persisted data so that the parent may be found at load time.

It is worth noting that the Storage classes are interfaces and may contain the data (e.g. in-memory storage), but they do not necessarily contain it, and in some cases absolutely do not contain it.

Root

Root is conceptually the location of a Repository and provides access to within-repository items such as a Mapper, a Policy, and the Registry. These provide further information about where actual Datasets can be found (currently with in-file-system repositories, the files are always stored under the folder indicated by Root).

TBD if Root will be a proper class, a named tuple, some other data type, or just parameters passed to an initializer in Butler, Repository, or other. It should contain basic bootstrap info for a Repository. It seems reasonable that the exact contents of Root should depend on the type of Storage being initialized.

Serialization

(todo)

Actions

Branch

Branch creates a new child repository.

Child configuration details TBD.

Probably a member function of Repository. Access via Butler(?) TBD.

Refactoring

1. Implement new Repository and Storage classes
    a. Can start only with PosixStorage. I think Gregory's multiple-storage request requires multiple inputs and multiple outputs, but this could be verified, and if only multiple outputs, the scope could be reduced.
    b. write unit tests and any additional needed proof of concept.
    c. from proof of concept refactor design and repeat step 1 as needed.
2. Replace Root use with new mechanism.
3. Replace direct filesystem access with Repository use. (only a few spots in Butler.py and in CameraMapper.py. TODO search github for any other uses in code.

Parent/Child configuration match requirement?

Should a repo be required to have the same configuration (mapper type, policy) as its parent(s)?

It seems like this could be problematic if a user wanted to:

- extend/add to the policy
    - e.g. add a template for a new output type
- change the policy
    - e.g. modify the naming template
- modify the behavior of the output mapper. An output repo should not have to have the same policy as the input:
- they each own their own mapper, all use of mapper and policy is contained within the Repository.

Conceptually, a parent is an input Repository to another Repository.

Ergo it seems like there is no technical reason to require a child to have the same configuration as its parent. From a user perspective this could be enforced at the API level but it does seem like there are reasons the user would want to have different Policy and/or Mapper at different levels of the hierarchy chain.

Repository Hierarchy and Configuration

When a repository with a parent is persisted it should create a persisted reference to its parent (exact implementation depends at least on storage type).

When a Repository class is instantiated from a persisted Repository it queries the Access for a parent repository. If there is one, the persisted reference contains enough information for the Repository to instantiate another Repository which is then stored as _parent. This happens recursively.

Configuration specification

Single Configuration

Creating a Butler with a Repo with a single config:

In the case where only a single repository (no parent or child) is configured:

```
repoCfg = Config(root="path/to/repo_2", mapper=SomeMapper, mapperArgs={...})
```

I think a repo could be instantiated by passing the config directly to Repository:

```
repo = Repository(repoConfig)
```

However I also think that more conventional use will be to pass the config to a Butler constructor and it will create the repository/repositories:

```
butler = Butler(repoCfg)
```

The Repository constructor will instantiate a single Repository, connected to the Storage at `root`.

Read Only or Read-Write Access

If the persisted repo at root is writable (not 'committed') then a writable repository will be created. If the repo is not writable then a not-writable Repository will be created. And if the repo becomes unwritable (is committed) during the lifetime of the Repository then calling `Repository.write` will raise an exception.

Parent Repositories

If a parent specifier is found in the persisted repository then the Repository constructor will create parent Repositories (that are kept as class members) as described in section Repository Hierarchy and Configuration

Multiple Configurations (specifying many parents/children at once)

This seems useful when:

- there are multiple repositories to be loaded into a single repo that has not yet been created.
- multiple children are to be specified at create time.

Flat vs. Nested

Problems with flat configuration

For example if config input paraemters are defined as:

```
Config(root, mapper, mapperArgs)
```

then only one config is defined in the object.

A parent config could be passed as:

```
Repository(config, parentConfig)
```

But if multiple parents (parent, grandparent, great-grandparent) then this API does not elegantly solve that. It could be done via a setParent method:

```
grandparent = Repo(config1)
parent = Repo(config2)
repo = Repo(config3)
parent.setParent(grandparent)
repo.setParent(parent)
```

Nested Configuration

It is more elegant to allow configs to be nested, and to allow Repository to traverse the nested hierarchy and create representative Repositories: e.g. with api: 'Config(root, mapper, mapperArgs, parentCfg)' you could say:

```
grandparentCfg = Config(root_1, mapper, mapperArgs)
parentCfg = Config(root_2, mapper, mapperArgs, grandparentCfg)
repoCfg = Config(root_3, mapper, mapperArgs, parentCfg)
repo = Repository(repoCfg)
```

It's a little cumbersome to respecify mapper and mapperArgs if they're going to be the same every time. Probably they could be assumed to be 'same as previous' unless explicitly stated. If loading from a persisted repo, this could be the case as well.

```
grandparentCfg = Config(root_1, mapper, mapperArgs)
parentCfg = Config(root_2, parentCfg=grandparentCfg)
repoCfg = Config(root_3, mapper=otherMapper, parentCfg=parentCfg)
repo = Repository(config)
```

At this point, it still takes 3 lines to set up the config which is (it seems to me) to be somewhat unpythonic, especially if much of the information (like mapper and mapperCfg) is repeated (IE same as parent) Config could have a method to add a dependent config, or a dependee.

```
Config(root, mapper, mapperArgs).parent(parentRoot).parent(grandparentRoot)
```

or

```
Config(root, mapper, mapperArgs).child(childRoot).child(grandchildRoot)
```

questions

- should a `childCfg` keyword argument be added to Config?
- what about aggregate repositories in nested config?

Multiple configs in the presence of persisted repository parent

What if a config is nested but the persisted Repository also contains a Parent Specifier?

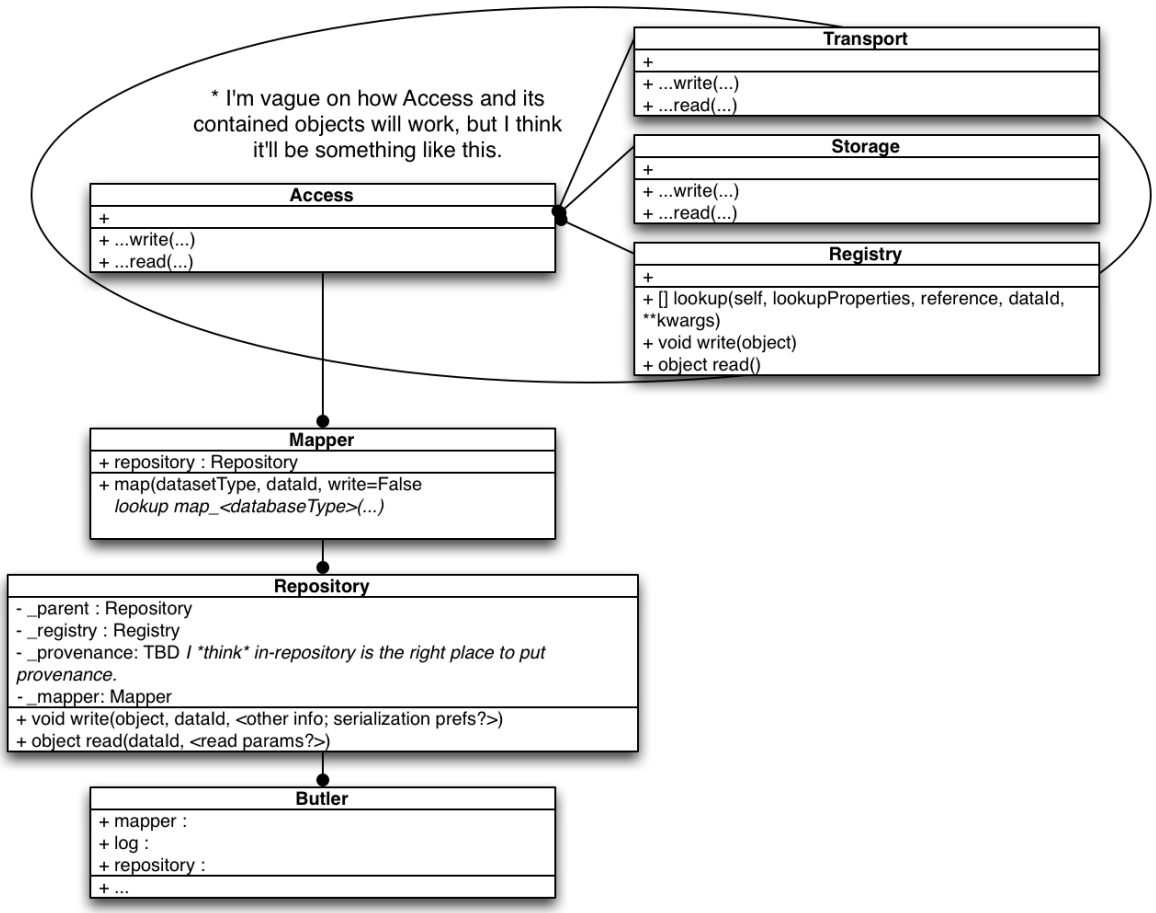
Options include:

- Raise an exception.
- Make it an aggregate repository.
- Make it a new branch; override the previous parent/child repo.
    - if it's a parent; will throw away any history of the previous repo. This is effectively similar to `git checkout --orphan`, I think.
    - if it's a child, then this effectively creates a new branch.

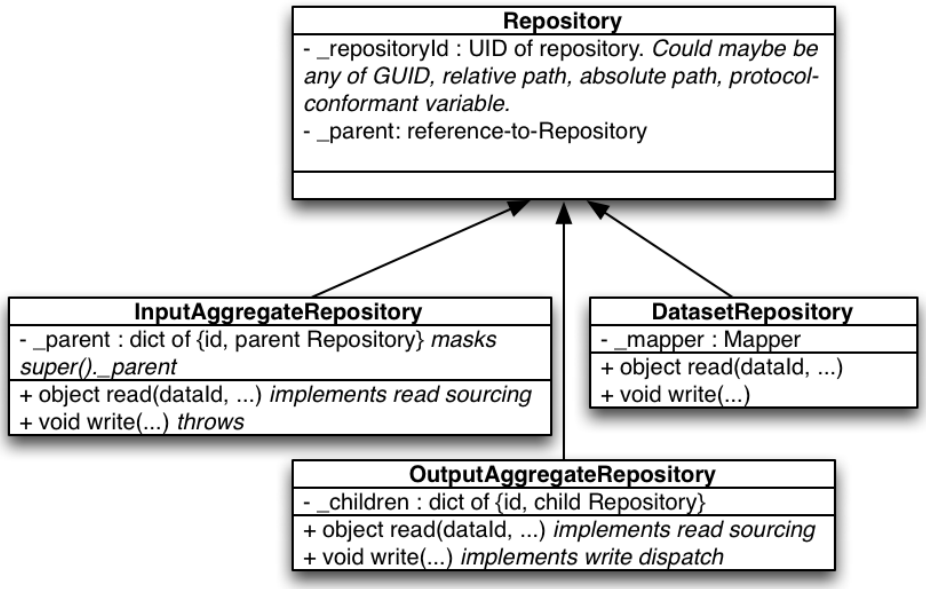
It seems like there are potentially cases where this behavior would be desired, and can be achieved by allowing new repositories to specify their lineage, but they would have to contain the entire configuration if they are to override the parent repo's persisted config info.

Diagrams

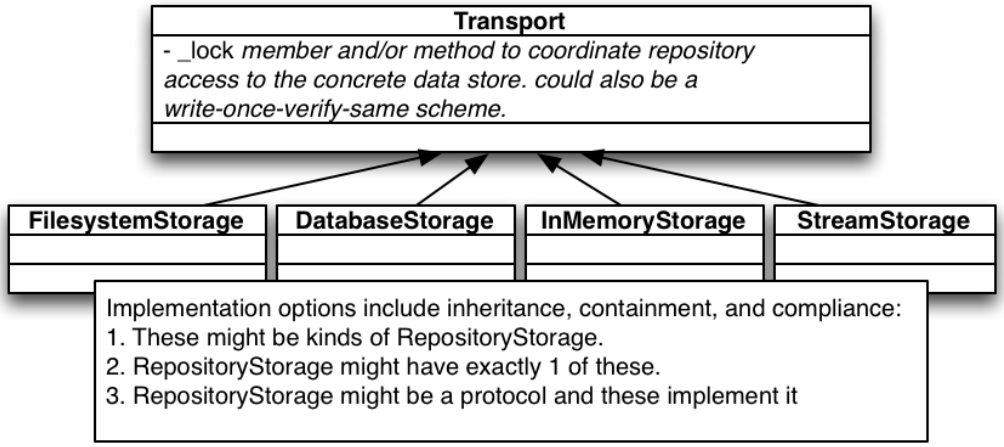
Butler Framework Class Diagram



Repository



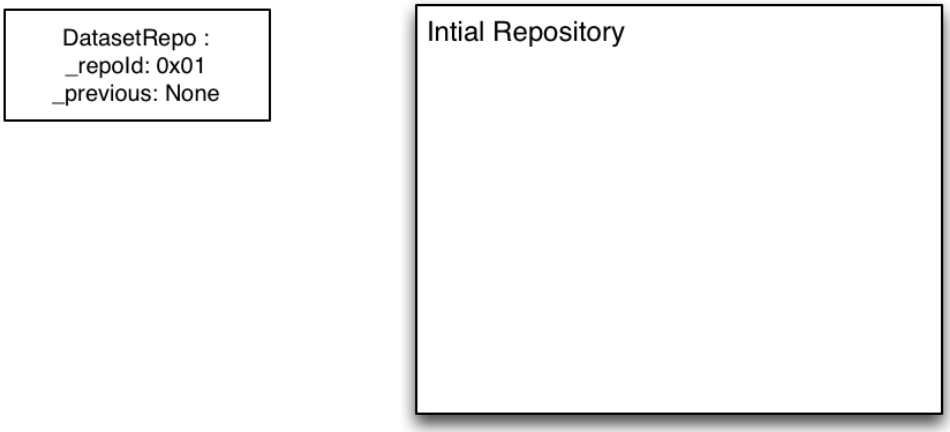
Storage



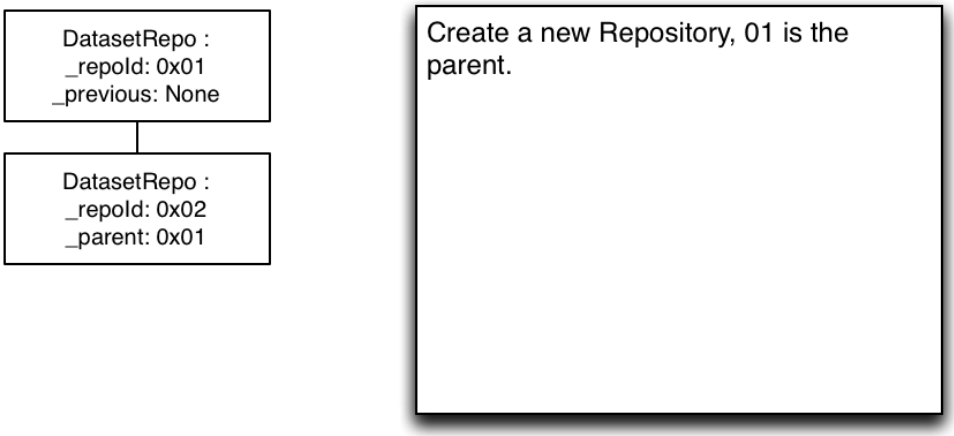
Parent and Branching

Relationship diagram of repository chain with Aggregate Repositories

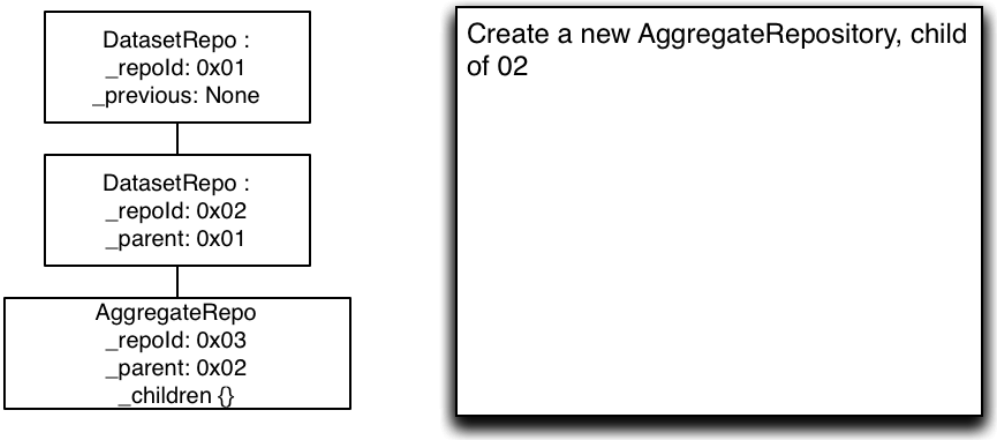
1



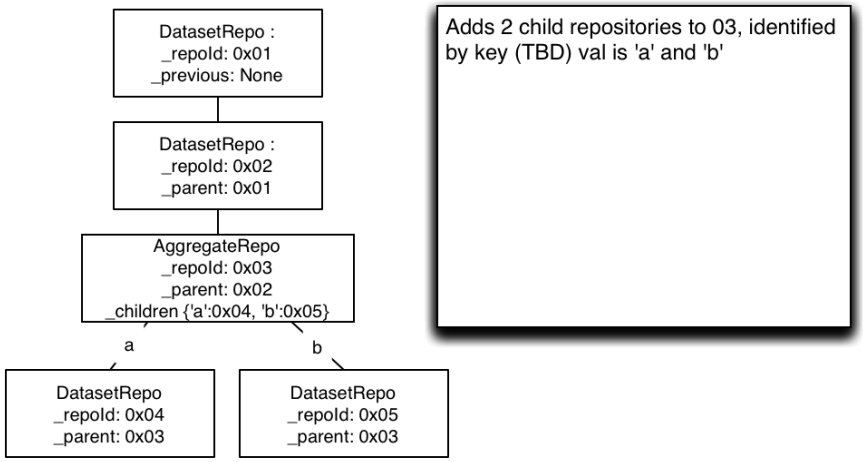
2



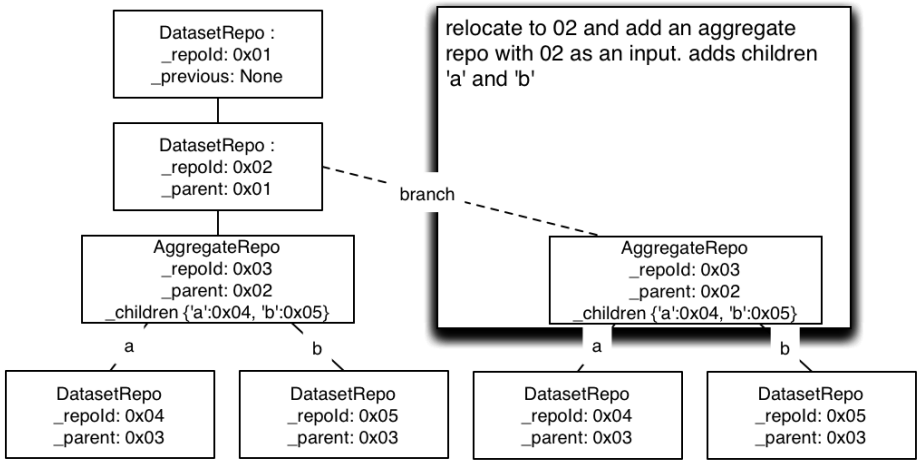
3


DatasetRepo :
_repoId: 0x01
_previous: None
DatasetRepo :
_repoId: 0x02
_parent: 0x01
AggregateRepo
_repoId: 0x03
_parent: 0x02
_children {}
Create a new AggregateRepository, child of 02

4


DatasetRepo :
_repoId: 0x01
_previous: None
DatasetRepo :
_repoId: 0x02
_parent: 0x01
AggregateRepo
_repoId: 0x03
_parent: 0x02
_children {'a':0x04, 'b':0x05}
a
b
DatasetRepo
_repoId: 0x04
_parent: 0x03
DatasetRepo
_repoId: 0x05
_parent: 0x03
Adds 2 child repositories to 03, identified by key (TBD) val is 'a' and 'b'

5


DatasetRepo :
_repoId: 0x01
_previous: None
DatasetRepo :
_repoId: 0x02
_parent: 0x01
branch
AggregateRepo
_repoId: 0x03
_parent: 0x02
_children {'a':0x04, 'b':0x05}
a
b
DatasetRepo
_repoId: 0x04
_parent: 0x03
DatasetRepo
_repoId: 0x05
_parent: 0x03
AggregateRepo
_repoId: 0x03
_parent: 0x02
_children {'a':0x04, 'b':0x05}
a
b
DatasetRepo
_repoId: 0x04
_parent: 0x03
DatasetRepo
_repoId: 0x05
_parent: 0x03
relocate to 02 and add an aggregate repo with 02 as an input. adds children 'a' and 'b'