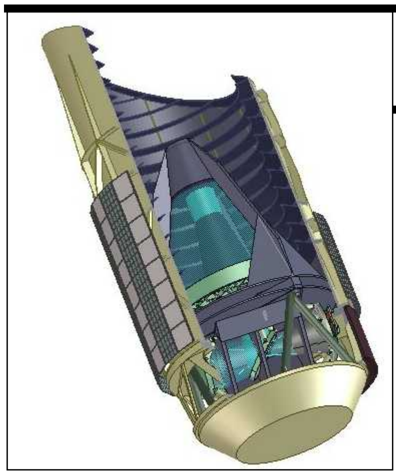


SNAP Design Document

Supernova Simulation Architecture

GB,AK,NK,GK

Revision	Date	Authors	Comments
1.32	8 August 2003	GB,AK,NK,GK	Last Change : Friday, September 12, 2003

Introduction and Overview

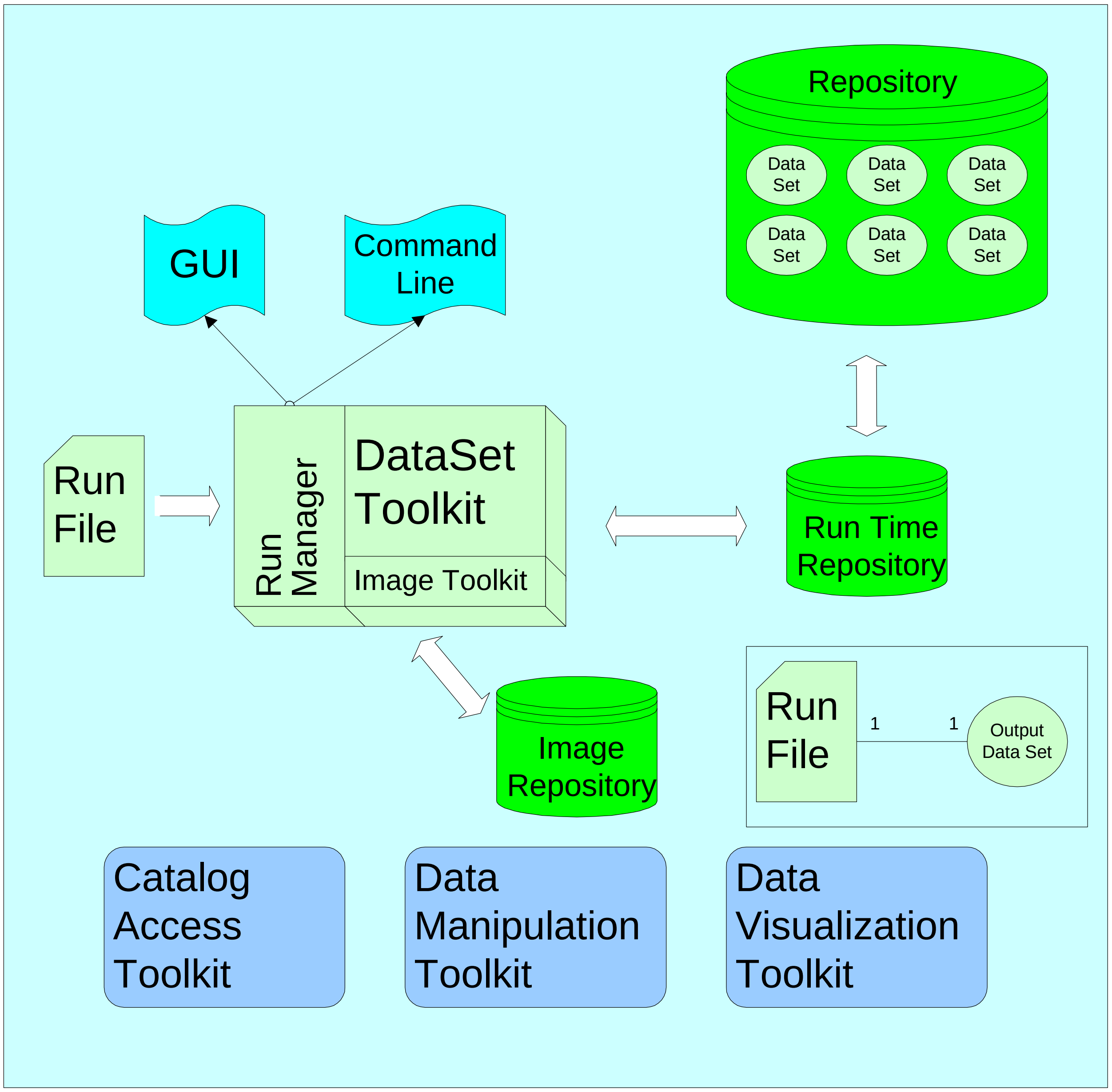
0.0 Preface

This system started from a set of requirements which led into a set of user scenarios and finally into a complete system. In this document, an overview of the system is presented first to give the reader some context. That is followed by the key requirements, a couple of user scenarios, the candidate APIs, some examples, and a section of definitions. The final section in the main document contains some specific rules that the system follows that a user who wishes to push the system might want to know. Following the main document are two appendices that will be of interest only to the people implementing the system.

Some readers may find it more understandable to read the document in a different order than that presented. Particularly some readers may be interested in reading the requirements or definition section earlier then presented.

Please note that since the different sections of this document were written over a span of time the nomenclature is not entirely consistent. That will be rectified in a later draft. In general, but not always, the Definition Section is the most up to date. Also, this document is still a working draft and contains some notes to the authors in red type. We apologize for these shortcomings and hope they do not cause too much confusion.

1.0 Overview



1.1 Overview

The primary goals of the SNAP simulation framework is to allow scientist users to run simulations and interact with data in as rich and natural a way as possible while allowing for the data to be validated and reproduced via a rich history mechanism. The fairly simple way we satisfied this requirements is by decomposing a simulation into *runs*. Some experiments have used the word *job* instead of run. The output of a run is a *DataSet* that contains the output results and everything necessary to trace and reproduce the results.

Examples of runs are a step in a process or a fork in an analysis. Each run is defined in a file and follows a specific format. Each run can access many datasets as input, but produces exactly one dataset as output. The output file contains everything necessary to trace and reproduce the results (History and Reproducibility). Repositories can be either an SQL database or a file system.

Examples of Runs :

- Run (A) could create a Universe. Run (B) could observe the Universe with SNAP. Run (C) could observe the same Universe with LSST. Run (D) could compare the SNAP and LSST observations of the Universe created in Run (A).
- Run (A) could be creating a light curve. Run (B) could be fitting the light curve.
- Run (A) could be one set of optics. Run (B) could be a modified set of optics. Run (C) could compare the effects of the modification.
- Run (A) could be one mission. Run (B) could be another mission

1.2 Run Manager, Data Set Toolkit and Image Toolkit

Throughout the design we've tried to use the following naming convention: Code that takes user supplied input and executes autonomously on that input is called a *Manager*; Code that provides a set of subroutines that user code calls is called a *Toolkit.*

The Run Manager reads a run file and executes the simulation. The DataSet Toolkit provides access to the datasets, and produces the output dataset. The Image Toolkit provides access to Image Files. There will eventually be both a GUI interface and a command line interface to the Run Manager. The command line interface will be scriptable with Jython.

1.3 Repositories

There are currently three repositories in the simulation architecture. The Data Repository stores DataSets and all other non-image data used in the simulations. It allows for reading, writing, and grooming. The Image Repository stores the images used in the simulation. It allows reading, writing, updating and grooming.   The Run Time Repository is used to keep track of and cache persistent objects that are in program memory.

The Image Repository allows updating because image processing requires many steps and saving all intermediary images would not be practical. Also, as long as the image manipulation process is saved, it is easy to reproduce the resultant image. The DataSets do not allow updating because doing so would make it difficult to support reproducibility. It is worth noting that the Data Repository is the same Repository describe in the Marseille Architecture document and can contain data that is not in DataSets and does not maintain a history. In order to insure reproducibility, Nouns must be saved using runs and stored in DataSets using the simulation toolkits.

The Run Time Repository is special in that there is no explicit user interaction with it. It is used to keep track of persistent objects that are in program memory in order to enable relationships to behave correctly as they move between data sets. Since Data Sets are write only and the SNOID of an object changes with each save, relationship pointers have to be updated to point to the correct logical object. The Run Time Repository insures that each time a user requests the same object from the persistent store she gets back the same Java reference. Also, as a final step to closing off a data set, the Run Time Repository uses its list of in-memory persistent objects to reconcile relationships.

1.4 Toolkits

The toolkits provide functionality that the can be used to during a run or at the command line to analyze a dataset. The toolkits will be a combination of SNAP specific code and existing libraries.

• Math Library Toolkit :	Provide scientific mathematical capabilities such as complex numbers and matrices
• Catalog Access Toolkit :	Access to shared lists of objects. *(still being worked on)*
• Data Manipulation Toolkit :	Standard data browsing and manipulation capabilities including export and import
• Data Visualization Toolkit:	Standard visualization toolkit

2.0 Requirements

There are many requirements, both implicit and explicit, that the system must satisfy. This section lists some of the most important ones that got refined and carried along as the system was designed.

2.1 Provide a convenient means for Nouns from one or more previous Runs to be restored into the program space.

2.2 Provide an ironclad traceability: sufficient information should be in a Run's output Dataset to allow complete reconstruction of the series of manipulations that created that Dataset (perhaps using other Datasets in the Repository). It is NOT currently a requirement that this reconstruction has to be done automatically, just that a human-traceable record must be present.

2.3 Allow Runs to be written in a modular fashion. This means that the author and user of a Run should only need to know about the existence of the Nouns in the Datasets that it will be using directly, it should not have to know the entire contents of the input Datasets to be useful. The Run code should only have to know about immediate predecessor Datasets, not their entire history.

2.4 Allow searching of all the DataSets in a repository. For example, to find which runs used the LSST telescope.

2.5 (We need to follow Relationships by Logical ID. The Logical ID used in this system is the human readable iName. A Supernova has a Link to a Galaxy (its host). The Galaxy undergoes some processing (maybe a photo-z determination); a later Run needs to know the photo-z for the host galaxy of our Supernova. We need to be able to find that Galaxy to get its photo-z. The State ID of the Galaxy has changed so it cannot be identified that way, we want to use a logical ID here. We may end up not actually using the Logical ID, but making the system behave as if we are. We're going to have to reconcile SNOIDs in any case and doing that could provide the proper function. This needs to be worked out.)

2.6 Want to be able to branch Nouns, meaning go off on different paths in which the Noun has different fates, for example re-observe a Universe with different read noise levels. State IDs will be different in each realization, so each is a "different" Noun in this sense, but logical ID's might be repeated in different realizations. If at a later point, the user want to work with multiple Nouns with the same Logical ID, it is up to the user to handle the ambiguity. In most cases the functioning of the State ID will happen under the covers and most users will never need to directly interact with the State ID.

2.7 Want to be able to use more than one data set as an input to a Run. Note this introduces possibility of 2 Nouns with same logical name but different contents/histories.

2.8 Need to be able to save composite Nouns (including the targets) to Dataset without explicitly saving all targets, so users can treat composite Nouns as single objects and don't need to know internal structure.

2.9 Want something simple, especially to start, because it's better and because we don't have time to write something complex.

2.10 Data Evolution Requirements *will go here* <Diagrams>

3.0 User Scenarios

The user scenarios document some of the ways that scientists would like to interact with the simulation framework irregardless of implementation details.

3.1 Detector Read Noise

Purpose : To determine what level of read noise will degrade the science. Need to figure out which statistics to use. Will need to interactively view the data.

Inputs:

- Need to specify what is the mission. There should be some standard missions defined that the user could use as is, or modify to suit.
- Need to specify which read noises to test.

Process:

- Loop over each read noise and compare W'.

User Experience :

- First the user needs to create a run. A run is defined as a simulation execution that is associated with a single input configuration, a single bit of simulation code, a single output data set and zero or more input data sets (*antecedent* data sets). The output set could contain many results if the code job produces them. There may someday be a GUI to create runs, or you could create the run as a file.
    - Creating the run as a file. The file format will be specified and there will be templates to follow.
    - The user chooses a name and puts that in the file. "Alex Kim Detector Test 52"
    - The next step is you have to specify a antecedent run if you're going to use data from a pervious run. This would be done by specifying the name of the run that you are using as an antecedent. There can be more than one antecedent, and the user must explicitly specify which one is being referenced. In this case, we'll use three antecedents: one that produced a universe, one that produced the observatory, and one that produced the mission. "GB Universe 32","SNAP Telescope 2","NK Mission Impossible".
    - (It is possible to access data from data sets that are not specified as antecedents. This is called *referencing* a data set. In that case, the accessed data is *imported* in to the simulation run. The imported data will not be saved unless the simulation code explicitly saves the imported data to the output data set. You would want to reference a data set as opposed to declaring it an antecedent if you want to bring in only specific Nouns with out having to worry about possibly introducing ambiguities. Always reference a data set if you only want to use a small subset of the data in it.)

    This is not a user experience and will be moved to a more appropriate location.
    - The next step is to create the run parameters. The run parameters are new nouns and verbs and simple java variables that are created new for this run. They are created in the User Configuration section of the run file as java. The reason the parameters are created in this separate section, and not the code section, is to aid other people in understanding, maintaining, and using the simulation.
    - The next step is to write your simulation code. See sample run file for example of code.
- The output file
    - Java version
    - Version of every noun and verb that gets loaded, saved or used
    - All non-deterministic input such as random number seeds.
    - Maybe three directories : all runfiles, output files and xml files share a directory

3.2 Produce Light Curves From a Ground Mission

Purpose : Make simulated light curves in order to test the fitting algorithm. Will need to be able to plot them.

Inputs : Universe, Supernova, Telescope, Mission

Process : The first step is to make an observing log. The observing log includes what days was the supernova observed, what were the conditions, and the configuration of the camera and exposure. Run the exposure calculator to figure out the signal to noise ratio for each exposure and from that realize the light curve.

User Experience I : Given a Universe with a Supernova (astronomical object) build its light curve.

- First the user needs to create a run. A run is defined as a simulation execution that is associated with a single input configuration, a single bit of simulation code, a single output data set and zero or more input data sets (*antecedent* data sets). The output set could contain many results if the code job produces them. There may someday be a GUI to create runs, or you could create the run as a file.
    - Creating the run as a file. The file format will be specified and there will be templates to follow.
    - The user chooses a name and puts that in the file. "Natalia LC Test 2"
    - The next step is you have to specify a antecedent run if you're going to use data from a pervious run. In this case, we'll use three antecedents: one that produced a universe, one that produced the observatory, and one that produced the mission. "GB Universe 32" (where GB Universe 32 has been populated with Astronomical Objects), "SNAP Telescope 2","NK Mission Impossible".

    - The next step is to create the run parameters. The run parameters are new nouns and verbs and possibly simple java variables that are created new for this run. They are created in a section of the run file as java.
    - The next step is to write your simulation code. See sample run file for example of code.
- The output file
    - Java version
    - Version of every noun and verb that gets loaded, saved or used
    - A copy or link to the runfile with includes expanded
    - All non-deterministic input such as random number seeds.
    - Maybe three directories : all runfiles, output files and xml files share a directory

4.0 Candidate APIs

Please note that these APIs are not yet documented. There are only sparse notes in "runfile v1.1.java". The most important information in this section is the Run File Template.

4.1 Run File Template

Run files are text files. The Run Manager uses the native java compiler built in to all Java run time environments to extract and compile the code and configure the DataSets that are antecedents or references. The preferred file extension for a run file is ".run".

```
Name : The name of the simulation run goes here.
Antecedents : List the Antecedent data sets here.
References: List any Referenced data sets here.

User Configuration
{
    Configuration data (initial conditions) that is
      particular to each run of this simulation goes here.
      Examples include the random number seeds, which model
      to use, etc.  This section is written in Java.

    For example, if a simulation might be run many times,
      while varying only certain parameters, those
      parameters should be initialized in this section.
      Also, if there are other parameters that define what
      is being simulated, those parameters should also go
      here.

    All configuration variables should be initialized in
      this section as a means of self documentation.
}

Program Code
{
    The simulation jave code goes here.  Most complicated
      code will be contained in Verbs or as methods of
      Nouns.  The code in this section can use all of Java,
      but will normally be primarily script like.
}
```

4.2 Run Manager

The runManager will probably have several UIs. For example there will be a command line version that can be used to integrate this system into eclipse so that run files can be compiled or run from eclipse.

```
String getRunName();
RID getNewRunID();
void run(runFile);
void compile(runFile);
```

4.3 DataSet Toolkit

```
void setSaveLevel(Default|Detailed|Debug|None);
```

```
DataSet getCurrentDataSet();
```

```
Noun importN(Class, iName, DataSet Name);
Noun importN(SNOID, DataSet Name);
Vector importNAll(Class, iName, DataSet Name);
void listUpdatedNouns(DataSet Name);
void listNewNouns(DataSet Name);
```

4.4 DataSet

A run has a Current DataSet that at the start of execution contains all of the Nouns from all of the antecedent DataSets. As the run progresses Nouns that the run saves are also stored to the Current DataSet. A run can retrieve any Noun that is in its Current DataSet by using the get() family of methods. When the run completes, the Current DataSet is called the Output DataSet and becomes read only.

A run can also access the Nouns in any arbitrary DataSet in the repository by *referencing* it. Once a DataSet is listed referenced section of the Run File the run can retrieve Nouns by using the inportN() family of methods. A Noun is imported only in to the run’s Java program space and not to its Current DataSet. If a run wants to save a copy of a Noun that it has imported, then it must explicitly save it to its Current DataSet.

A user should declare a DataSet to be an antecedent if most of the Nouns in that set are useful to the current run or might be useful to a future user of the Output DataSet. A user should declare a DataSet to be a reference if there is only a very specific and limited set of Nouns that the user wants use and they do not want to pollute the Current DataSet’s namespace with a lot of unused data. As an example consider the case where there is one complete simulation containing many Nouns that were used though all phases of the simulation. Import should be used if the user wants to just grab the Universe that was used without bringing along every supernova measurement, light curve fit, etc. If the user specified the previous simulations DataSet as an antecedent then there would be many confusing objects with similar iNames in the Current DataSet before the new simulation even begins.

```
void save(Noun);
void saveDetailed(Noun);
void saveDebug(Noun);
Noun get(Class, iName);
Noun get(SNOID);
Noun get(Class, iName, DataSet Name);
void delete(SNOID);
void delete(Class, iName);
Vector getAll(Class, iName)
Vector getAll(Class, iName, DataSet Name);
Vector getAll(Class)
Vector getAll(Class, DataSet Name);
void close()
```

4.5 Image Toolkit

```
Image(Name, Size);
Image get(Name);
Image getCut(Name, cut);
Image get(DiskImage, boolean CRC_MUST_MATCH);
Image getCopy(Name);
save(Image);
SaveNew(Image, Name);
DiskImage getDiskImage();
```

```
Image is not a Noun.  Each Image has a DiskImage which is a
      Noun.

DiskImage.getName()
DiskImage.getCRC()
```

4.6 Catalog Access Toolkit

```
/**
 * Community Resources : Catalog of information objects.
 * Catalogs do not
 * use the history and reproductivity framework.
 */
CatalogManager : Interface {
    Catalog getCatalog(Name);
    Catalog copyCatalog(Name);
    saveCatalog(Catalog, Name);
}

Catalog :
    closeCatalog(boolean) : Make read only
    Catalog getAllFromCatalog(Class)
    Catalog getAllAOFromCondition(functor)
    Noun getFromCatlog(SNOID);
    Noun getFromCatalog(iName);

    CopyObjectTocatalog(Noun)
    updateObject(Noun)
```

4.7 AutoMerge Verb

The AutoMerge verb takes DataSets that have branched and recombines them in to a single DataSet. This is useful if different operations on the same objects are calculated in distinct non-linear operations. The AutoMerge only works if the branching follows a fairly simply topology. If there are ambiguities that the following rules can't resolve then the AutoMerge Verb will throw an exception and it is up to the user to manually resolve the ambiguity by writing a custom Merge Verb. Writing a custom Merge Verb is in general an easy task.

The AutoMerge Verb uses the following rules :

- Objects with the same iName become the same object. Another way to say this is that objects with the same iName are (assumed to be in this case) logically the same object, and after this verb is run they become actually the same object.
  - Fields that are not the default value or equal in more than one object with the same iName will be merged. It is an error to have more than one object with an iName that have non-default, non-equal values for a field.
  - The purpose of this operation is to allow different fields of an object to be calculated in different simulation runs and then combined back in to one object.
- Links are reconciled by iName
  - When the objects are coalesced by iName in the above step, any object that pointed to any of the pre-coalesced distinct objects will point to the one final coalesced object.
- The procedure implies that Nouns are written using a testable non-valid default value.

5.0 Examples

5.1 Run File Example I

```
Name : Alex Kim Wprime 52
Antecedents : Universe 5, Mission NK Impossible
referenced  : SNAP Telescope 3

User Configuration
{
    double readNoise[3] = {10.4, 15.3, 22.4};

    double randomSeed = 52;

    DataManager.setSaveLevelDataManager.Default);
//  DataManager.setSaveLevelDataManager.None);
//  DataManager.setSaveLevelDataManager.Detailed);
//  DataManager.setSaveLevelDataManager.Debug);
}

Program Code
{
    //  Get current DataSet.  The DataSet lets the user read from any of the
    //  antecedents or referenced as well as save objects.
    DataSet ds = DataManager.getDataSet();


    Universe  U = ds.get(Universe.Class, Universe.True);
    Mission   M = ds.get(Mission.Class, "Impossible");
    Telescope T = ds.importN("SNAP Telescope 3", Telescope.Class, "Telescope");

    //  If more than object with the same iName is in the data set, trying to
    //  get() that iName will throw an exception.  In order to access the
    //  objects you may use the getAll() method.
//  NounArray  v = mi.getAll(Supernova.Class)

           //  The other method to get an object with an ambiguous iName is to specifiy
           //  which data set to get the object from.  This, of course, only works if
           //  the specified data set has only one object of that class with that
           //  specified iName.  In this case, it is a good idea to change the iName
           //  so that people who use the output data set won't have the same issue!
//  Universe  U = ds.get(Universe.Class, Universe.True, "Universe 5");

    Colt.setSeed(randomSeed);

    for (int i=0; i<readNoise.Length; i++) {
        T.setDectorNoise(readNoise[i]);

        CosmologyAnswer ca = CalcCAnswer.run(U, T, M);
        ca.setiName("RN:" + readNoise[i]);
        ds.save(ca);

        //  debug or intermediate steps can be saved in such as a way that the
        //  caller can determine if they will be saved to the output.
    //  ds.saveDetailed(ca);

    //  ds.saveDebug(ca);
    }
}
```

5.2 Run File Example II

```
Name : Natalia Light Curve 2
Antecedents : Universe 5, Mission NK Impossible, SNAP Telescope 3
Referenced  :

User Configuration
{
    String supernovaName = "2010A";

    double randomSeed = 54;

           DataManager.setSaveLevel(DataManager.Default);
}

Program Code
{
           //   get output DataSet
    DataSet ds = DataManager.getDataSet(); // get output data set

    //  Universe.True = "True"
    Universe  U = U5.get(Universe.Class, Universe.True);
    Telescope T = uc.get(Telescope.Class, "SNAP");
    Mission   M = mi.get(Mission.Class, "Impossible");

    Supernova S = U.getAstonomicalObject(supernovaName);

    Colt.setSeed(randomSeed);

    //  Build observation log for entire mission.  Vector of Observations
    NounList ol = BuildObservationLog.run(T, M);

    //  Find the observations of this one object
    NounList snol = FindObservationLogFor.run(ol, S);

    LightCurve lc = new LightCurve();
    lc.setName(S.getName());

    for (int i = 0; i < snol.length(); i++) {
        //  SignalToNoise include signal and noise
        SignalToNoise sn = CalculateSignalToNoise.run(snol.at(i), S);
        lc.add(sn);
    }

    ds.save(ol);
    ds.save(lc);
}
```

5.3 Verb Code Example

```
public final class CalculateSignalToNoise
```

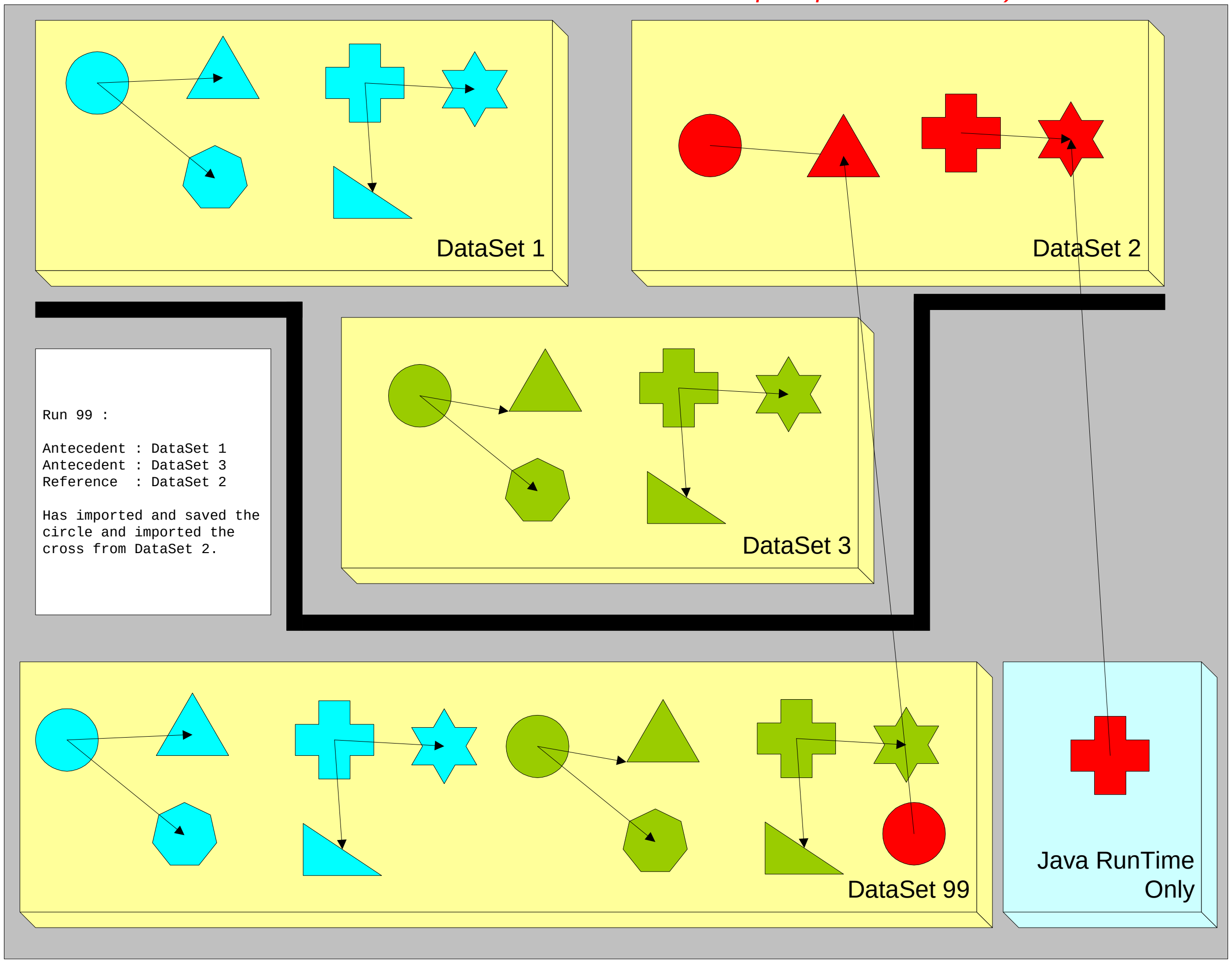
```
extends Noun implements Verb
{
    public static final String
    getName()
    {
        return "Signal To Noise Calculator by Natalia";
    }

    public static final String
    getVersion()
    {
        return "%%Version";
    }

    /** Nouns need this **/
    public static short getClassID()
    {
        return 4;
    }

    // Meta Data code code goes here
    static {
        // DDL
    }

        //---  Noun Parameters
    private double flatFieldError = 0;

        //---  Noun Methods
    public void
    setFlatFieldError(double anError)
    {
        flatFieldError = anError;
    }

    public double
    getFlatFieldError()
    {
        return flatFieldError;
    }

    /**
     * This method allows the verb to be called outside the simulation
     * framework where DataSets have no meaning.  DataSet.NOP is defined with
     * in the DataSet class to look like a DataSet but each method is a NOP.
     */
    public final static SignalToNoise
    run(Observation theObs, AstronomicalObject theAO, Universe theUniv)
    {
        run(DataSet.NOP, theObs, theAO, theUniv);
    }


    /**
     * Calculate Signal to Noise given a Observation and an AstronomicalObject.
     * These are bad comments aren't they!
     *
     * We are ignoring any host galaxy !
     *
     * @param ds        the DataSet
     * @param theObs    The Observation
     * @param theAO     The Astronimical Object
     * @param theUniv   The universe
     *
     * @return An instance of SignalToNoise for the given parameters.
     */
    public static final SignalToNoise
    run(DataSet ds, Observation theObs, AstronomicalObject theAO,
        Universe theUniv)
    {
        //  Telescope here is a specific realization of the telescope at the
        //  moment of the observation, so it has a configured camera.
        Telescope T = theObs.getTelescope();
        Camera    C = T.getCamera();
        CameraConfiguration CC = C.getConfiguration();

        //  Nouns can be optionally saved such that during debug or analysis
        //  they are readily available.  Usually a verb only saves Nouns using
        //  saveDebug() or saveDetailed();
        ds.saveDebug(T);
        ds.saveDebug(C);
        ds.saveDebug(CC);

        //---  First find the signal

        //  Calculate the flux at the observatory (above atmosphere)
        Flux inputF = U.CalcFlux(theObs.getJD(), S);

        //  Get Optical Elements from the telescope.  The Noun
        //  CompondOpticalElement contains an ordered set of OpticalElements
        //  and can perform operations on them
        CompoundOpticalElelement coe = T.getOptics();
        coe.append(new OpticalElement(theObs.getTransmission));
        OpticalElement oe = coe.getEquivalent();

        //  Since there is no ambiguity to the procedure, we make this a
        //  method instead of a verb.  If there would be more than one
        //  possible way to do this, it should be a verb!
        Flux whiteF = oe.PropogateFlux(inputF);

        Filter filter = CC.getFilter();

        Flux filteredFlux = filter.PropogateFlux(whiteF);

        //---  Now calculate the noise

        Flux skyWhiteFlux = theObs.getWhiteSkyFlux();

        coe = T.getOptics();
        coe.add(CC.getFilter());
        oe = coe.getEquivalent();
        ds.saveDetailed(oe);

        Flux filteredskyFlux = oe.PropogateFlux(skyWhiteflux);

        //---  Now take into account camera

        double readNoise   = C.getReadNoise();
        double darkCurrent = C.getDarkCurrent();
        double exposure    = theObs.getExposure();

        double s2n = ExposureTimeCalculator.run(readNoise, darkCurrent,
                                                exposure, filteredFlux,
                                                filteredskyFlux);

        Flux observedFlux = filteredFlux + Random.gauss(filteredFlux / s2n);

        //  takes true observed signal, true signal, and noise
        SignalToNoise returnS2N = new SignalToNoise(observedFlux, filteredFlux,
                                                    filteredFlux / s2n);

        ds.saveDebug(returnS2n);

        return returnS2N;
    }
}
```

5.4 Noun Example

5.5 DataSet Evolution Example I

In this example runs 1, 2 and 3 have already run and produced DataSets 1, 2 and 3. Run 99 is now running and the diagram is a snap shop of the Current DataSet for run 99. Run 99 has declared DataSets 1 and 3 as antecedents and DataSet 2 as a referenced DataSet. At this point in the run, the circle from DataSet three has been imported <u>and</u> saved to the Current DataSet and the Cross as been imported but not saved.

It is important to note that when importing an object, the Links that that object has are maintained and can be followed. This functionality is free given the current underlying design. This allows the user to, for example, import a Universe and save it so it can use, and then follow the appropriate Links and save those Nouns also for future use. (*Although to avoid confusion we may wish to change the system so that Links are explicitly broken and therefore can not be followed. We can leave this as an open question for now.)*



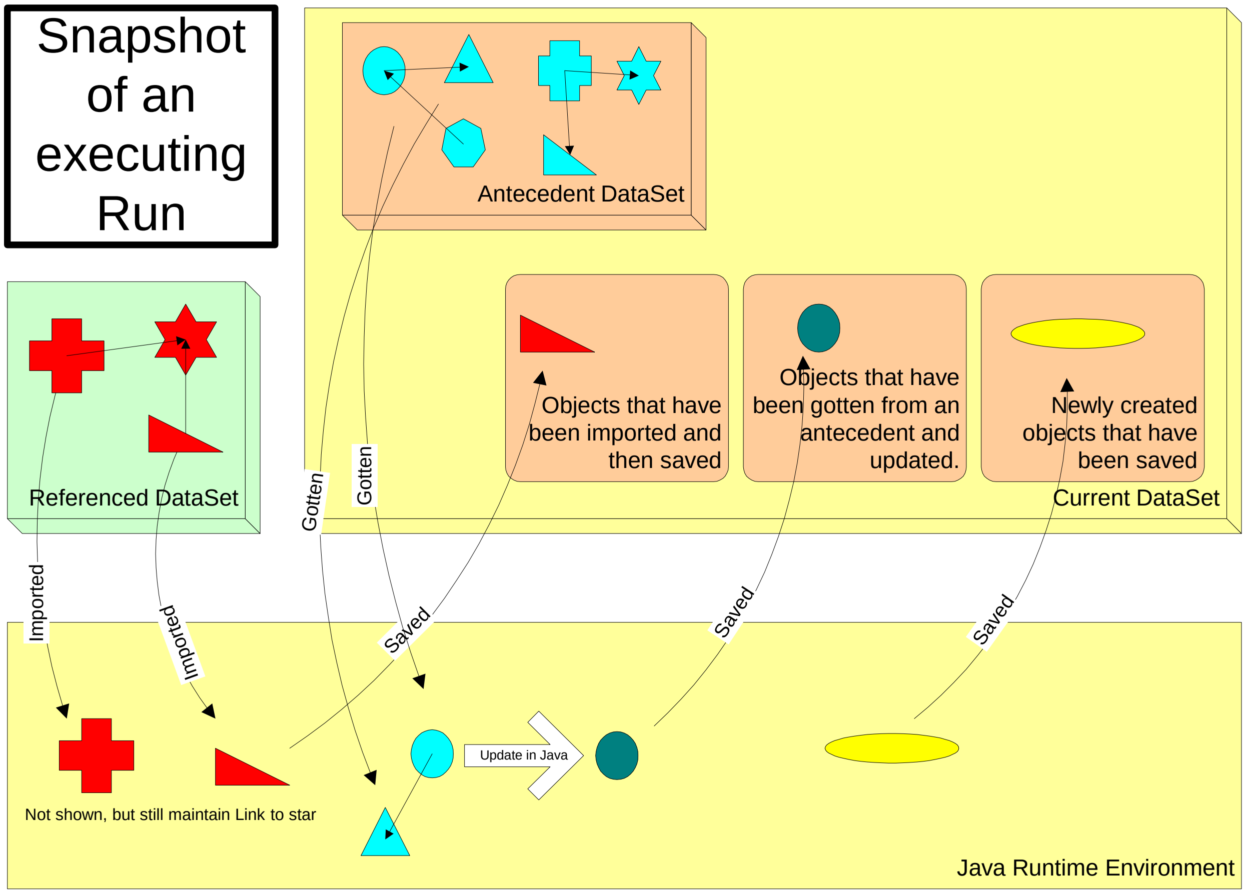
```
Run 99 :

Antecedent : DataSet 1
Antecedent : DataSet 3
Reference  : DataSet 2

Has imported and saved the
circle and imported the
cross from DataSet 2.
```

5.6 DataSet Evolution Example II

In this example we take a two snapshots of a DataSet. The first snapshot is while a run is executing and DataSet is read-write and defined to be the Current DataSet of that run. The second snapshot is after the execution has completed and the DataSet is now read-only and defined as an Output DataSet of that run.

5.6.1 During Execution : The Current DataSet

Here is the DataSet as the run is executing. The Jave Runtime environment is the program space where the simulation is running. At this point in the execution, the simulation has imported the cross and the red triangle from a referenced DataSet. It has also gotten the circle and the blue triangle from an antecedent DataSet. Since the blue shapes are in an antecedent DataSet they are now in this run's Current DataSet and can be gotten without specifying which DataSet they came from. The simulation has modified the blue circle and also created a yellow oval and saved those to its Current DataSet.



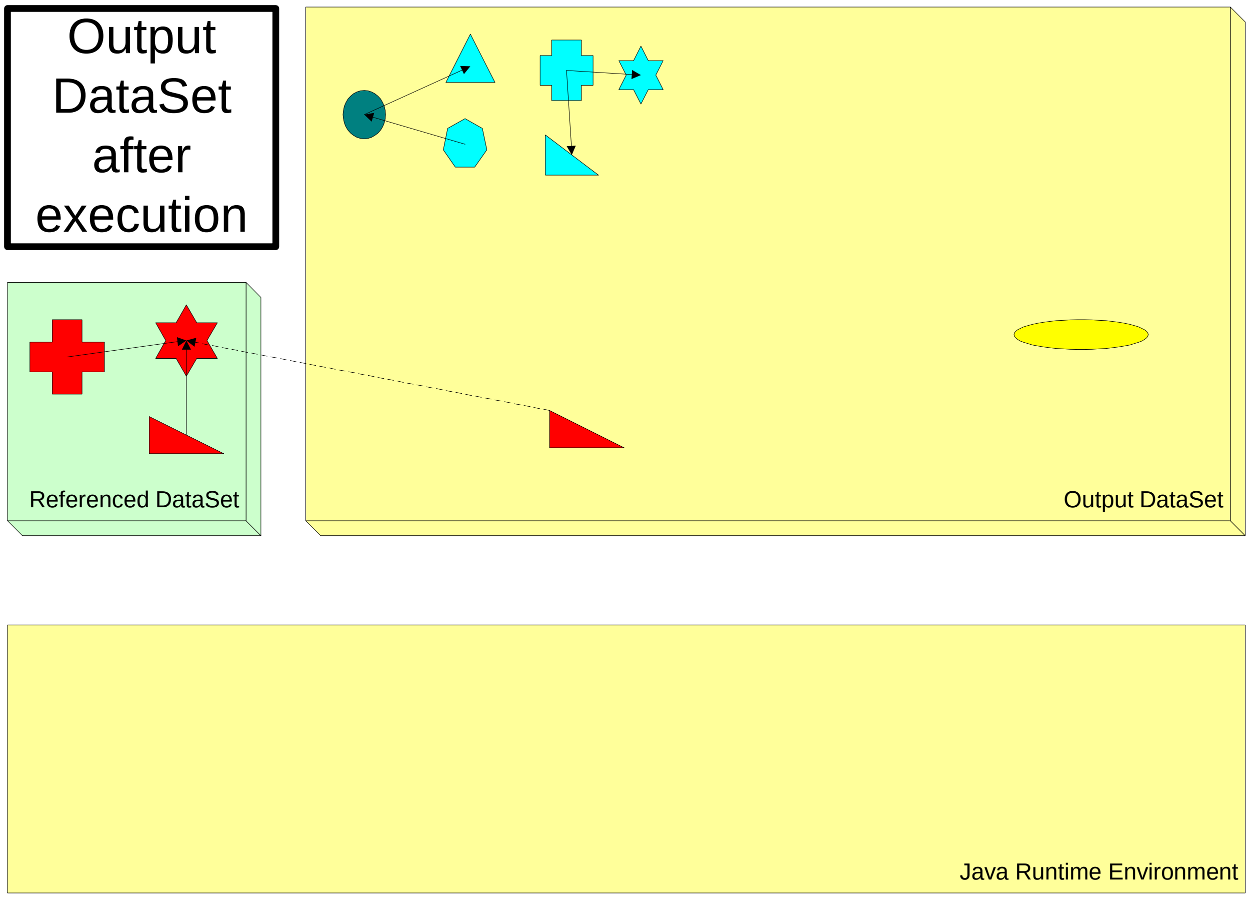
Snapshot of an executing Run
Antecedent DataSet
Objects that have been imported and then saved
Objects that have been gotten from an antecedent and updated.
Newly created objects that have been saved
Current DataSet
Referenced DataSet
Gotten
Gotten
Imported
Imported
Saved
Saved
Saved
Update in Java
Not shown, but still maintain Link to star
Java Runtime Environment

5.6.2 After Execution : The Output DataSet

Here is the DataSet after the simulation run has completed. The DataSet has been closed and is from this point forward read-only. Although the information is available through the History mechanism, any distinction of where an object originally came from is lost. All objects are now just in the Output DataSet.

The updated blue circle maintains the correct Links and has overwritten the previous blue circle. Note that the hexagon that pointed to the original blue circle, now points to the updated blue circle. This is the expected behavior and occurs even though the new blue circle has a different SNOID than the original blue circle.

The red triangle was imported from the referenced DataSet and saved into the Current DataSet so it is now in the Output DataSet. Note that it maintains its link to the red star even though the star is in a different DataSet. The simulation framework has the capability to follow Links across DataSets in the case of importing objects.



6.0 Coding Idioms

6.1 How a verb should save an intermediate result

When writing a verb if there is an intermediate result that might be of interest to the caller, it should be saved using the saveDetailed or saveDebug method.
Depending on how the caller of the verb configured the DataSet, saveDetailed or saveDebug may or may not actually save the verb. By using these methods, the writer of the verb allows the caller to determine what level of intermediate results should be saved. saveDetailed

should be used for most intermediate results, and saveDebug should be used when the intermediate results will only be useful when the most rigorous checks are being done.

The DataSet currently has three settings :

- save() gets saved and saveDetailed() and saveDebug() are ignored
- save() and saveDetailed() get saved and saveDebug() is ignored
- save(), saveDetailed() and saveDebug() are all saved.

If needed more levels might be added at a future date.

6.2 How a verb's run method should be coded

In order to maximize the odds that Verb code will be compatible with future framework evolution, the verb run method should be coded as follows :

```
return_type run(DataSet ds, parameter_list)
{
    verb_code
}

return_type run(parameter_list)
{
    run(Directory.DATASET, parameter_list
}
```

Following this template allows the verb to be used in the simulation framework that uses DataSets or the lower level Marseille Architecture that uses the Directory. Whatever future architecture we use for the mission and analysis pipelines will probably be based on something like the Marseille Architecture.

Directory.DATASET is a special DataSet that acts as a limited functionality proxy for a particular Directory database and file.

7.0 Definitions

7.1 RUN: a single invocation of the simulation framework, e.g. one run of a Run file.

7.2 REPOSITORY: The entirety of the persistent data available to a Run, i.e. the observable universe of persisted data. Could be everything in a single database, everything in one directory or on one machine, etc. There can be more than one Repository, but only one at a time can be used for a given Run. A Repository contains DATASETS and also other persistent data which are not DATASETS, i.e. not part of a simulation.

7.3 RUN TIME REPOSITORY: The Run Time Repository keeps track of all of the persisted Nouns that have been retrieved in to the Java program space.

7.4 CURRENT DATASET: During a run there is one DataSet that is directly associated with the run. All Nouns that are saved by the run are saved to the Current DataSet. Also, any Nouns that are in a run's antecedent DataSets are placed in the Current DataSet at the beginning of the run. During the execution of a run, all of the Nouns that are in the current DataSet either because the run has already saved them or because they were in an antecedent DataSet are available to be gotten (get()). When a run is complete, the Current DataSet becomes the Output DataSet.

7.5 OUTPUT DATASET: The persistent data that is the output of a Run. It lives inside a Repository, which may hold multiple Datasets. One Run produces one and only one Dataset on output. A Run can access more than one Dataset on input, however.

7.6 ANTECEDENT: If a DataSet is an antecedent to a run then all of the Nouns in that DataSet are put in to the new run’s DataSet.

7.7 REFERENCE: If a DataSet is referenced by a run, then the Nouns in that DataSet are available to be imported into a run’s Current DataSet.

7.8 NOUN: All data intended to be saved or restored to/from Datasets must inherit from this base class. A NOUN can have methods associated with it, but by convention those methods should always be closely tied to the definition of the NOUN. Examples would be setting a parameter of the NOUN. For a precise definition of a Noun please see the Marseille Architecture document.

7.9 VERB: Code that is not tightly coupled to the definition of a NOUN should implement the VERB interface. VERBS can call other VERBS and can be called directly in simulation code. If a VERB has settable parameters it should also inherit from the NOUN base class so that those parameters can be saved in the DATASET. For a precise definition of a Verb please see the Marseille Architecture document.

7.10 RUN MANAGER: The code which manages the running of simulation Runs.

7.11 RELATIONSHIP: a member of a source Noun that indicates an association with another Noun. The former is the source, the latter the target. Can be Composition or Link.

7.12 COMPOSITION: Type of Relationship indicating that the target is wholly owned by the source. A Noun can be the target of only one Composition, but the source of any number. When the source is destroyed, the target is destroyed with it. Source is a "composite Noun." A Composition is analogous to a compound class in C++ or Java.

7.13 LINK: Type of Relationship that indicates the target exists outside the source. A Noun may be the source and/or target of any number of Links. A Link is analogous to a pointer in C++ or a reference in Java.

7.14 LOGICAL ID: An identifier for a Noun that persists through changes to the Noun. We have an "iName" that we will serve this purpose.

7.15 STATE ID: An identifier that changes whenever a Noun is saved to the Repository with any alteration from its previous state. We have a "SNOID" that will serve this purpose. We will assume it to be a rule that any two Nouns in the Repository sharing a SNOID are the same Noun, i.e. this is a globally unique identifier in the Repository. In most cases whatever happens with a SNOID will happen behind the scenes and users will seldom if ever directly interact directly with a SNOID. Users will generally refer to objects using the Logical ID (iName).

7.16 PROGRAM SPACE: Instantiated Java objects in the Run. Once the Nouns have been retrieved from the Repository, they exist within the accessible memory as Java objects and are manipulated by the language constructs, not by the DataSet Toolkit. A Noun in the program space can be changed without acquiring a new SNOID for every change, and Nouns exists in the program space that need not be known to the Current DataSet.

8.0 System Notes for Advanced Applications

8.1 When a modified object is saved it gets a new SNOID.

8.2 There are methods in the DataSet Toolkit which allow a user to see which Nouns are new to that DataSet and which Nouns have been updated in that DataSet.

8.3 When a modified object is saved and gets a new SNOID, all objects <u>in the Current DataSet</u> that pointed to the unmodified object must point to the new object. In other words, within a data set, logical relationships must hold as Nouns are updated.

8.4 iNames are the Logical IDs. They are assigned by the clients and not guaranteed to be unique. Users can change the iName of a Noun, there is no saving of old names or any means of reconstructing history.

8.5 SNOIDs are the State IDs. Two Nouns with the same SNOID are guaranteed to be the same object.

8.6 A Noun is considered altered if any of its Composition targets are altered, but not if a Link target is altered.

8.7 Datasets are write during the run and then become read-only.

8.8 A Run can designate any number of Datasets as Antecedents. Once a Dataset is an Antecedent, any Noun in the Antecedent is considered to be part of the current Dataset and will be automatically become part off the output DataSet. Thus Output Dataset is a cumulative collection of all Nouns in all of the Antecedent DataSets. Note that this is a logical behavior and doesn't mean that each Dataset will contain a complete representation of each Noun.

8.9 A Run can import() a Noun from any Dataset. A request returns a Java reference to the reconstructed Noun in its Program Space.

8.10 A get() or import() specifies the following:

8.10.1 The class or interface desired. Any derived classes or implementations are valid matches.

8.10.2 The Dataset from which the Noun is to be extracted. The current Dataset is used for get(). Note that all the Nouns of the Antecedents are considered to be in the Current Dataset.

8.10.3 An optional iName of the desired Noun. get() throws exception if more than one Noun of the desired class and iName exists in the target Dataset. In that case, the user should use getAll().

8.11 There is also a getAll() method which returns a container of Nouns. For now getAll() takes the same arguments as get().

8.12 All of the data in a run's Antecedent Datasets are, at least logically, also contained in the run's output DataSet.

8.13 There is a way to groom or "prune" the history tree of undesired DataSets.

8.14 A user can delete objects from the Current DataSet only. This is an advanced operation and should only be done with great care. The reason a user might want to delete an object is to resolve an ambiguity.

8.15 When the Run finishes, its Output Dataset gets put into the Repository for use by other runs.

8.16 Part of each dataset is the Run File of the Run that created it. The Run File includes all arguments, antecedents, and any non-deterministic quantities, namely random number seeds.

9.0 Appendix I : Open Issues

9.1 There was a suggestion that it would be good to do a ds.saveDetailed() on the input parameters of a Verb.

This seems like a good capability, but the system as designed doesn't offer a great way to do this. Here is the thinking so far:

If we save the input parameters, then we're saving the calling Nouns that were passed in and maybe the caller doesn't want them saved for some reason. Right now, there's no way to differentiate who saves objects and it isn't sequential like a history log. We might want to add something like that, but it wouldn't be a trivial change, although it is unclear if it would raise the level of difficult—we might be able to add some sort of sequential debug log of Nouns. Right now we it is in the plan to have a text history log.

10.0 Appendix II : Required Changes to Marseille Architecture

10.1 SNOID

The SNOID will need to change in the following ways :

- Double the size of the SNOID
- Increase the Class ID field in the SNOID so it can handle many more classes
- Add a new field called Run ID

There is an issue that with the SNOID so large it will sometimes be larger than the data it names!

10.2 Verb

10.2.1 Idioms

Verbs should be implemented using the following idiom to allow them to be used both in the simulation framework and the future analysis framework.

```
Verb.run(param1, param2)
{
    run(DataSet.NOP, param, param2);
}

Verb.run(DataSet ds, param1, param2)
{
    // code goes here
}
```

10.2.2 Structure

Verbs once again become singletons and the run methods become static.

10.3 Nouns

10.3.1 PerviousSNOID

Nouns need a new field : previousSNOID

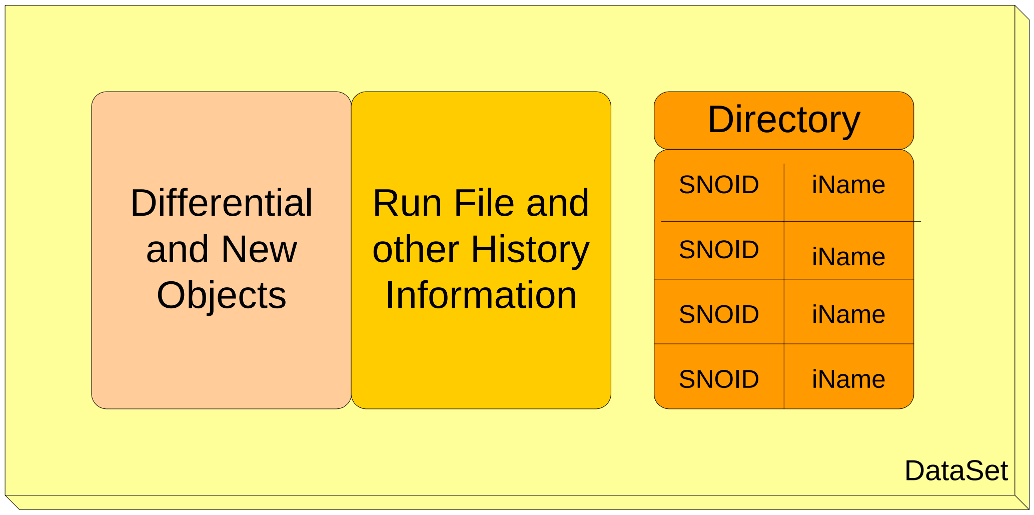
10.3.2 Default and Null Values

Noun fields in Nouns need to be able to support the values Null and Default.

11.0 Appendix III : Implementation Notes

11.1 DataSets and Antecedents

Since DataSets logically contain all the antecedent Nouns, the data structure for DataSets should be efficient and not need to store unchanged Nouns in each DataSet. Since SNOIDS are unique within a repository, here is one possible implementation.


Differential and New Objects
Run File and other History Information
Directory
SNOID
iName
SNOID
iName
SNOID
iName
SNOID
iName
DataSet

The Directory could contain only the Nouns stored in other DataSets or it could store all the Nouns including the differential and new Nouns.