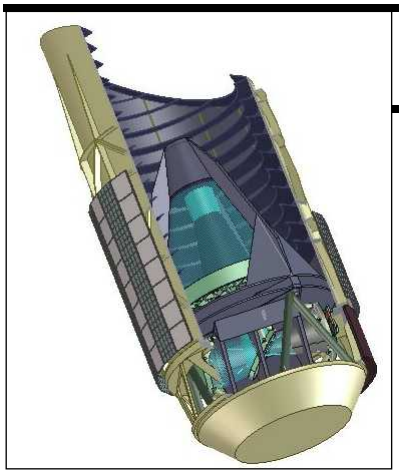


SNAP Design Document

The Marseille Architecture

GB,WJ,AK,JL,SM,GK

***Please Note** : In its current form, this is a working document and is primarily for the benefit of the Authors and core design team. At a later date, the contents will be transformed into an appropriate document for wider distribution Please Also Note : This document is out of date in many sections.*

Revision	Date	Authors	Comments
0.70	22 July 2003	Kushner	Last Change : Wednesday, November 05, 2003

1.0 Task List

1.1 Configuration Files
1.2 Visualization Tools, Histograms
1.3 Data Browsers
1.4 Matrix Arithmetic (Colt)
1.5 Vector Arithmetic

Pull a column or data field out of a collection. Operate on it. Put it back. (from FNAL)
Note: Could be implemented as an action that takes out + an action that puts back.

1.6 FITS I/O Library (Goddard?)
1.7 Complex Numbers

2.0 Introduction

3.0 Nouns and Verbs : An Overview

(How to Use overview goes here)

Nouns are shared object classes that can be saved to a data store. Specifically, Nouns are java classes that extend the class Noun and follow a few rules. Examples of Nouns are: AstronomicalObject, Supernova, MeasuredLightCurve, Cosmology, and Mission.

In the definition of Nouns, the word “shared” should be interpreted broadly. For example, instances of Nouns are used to share, or pass, information around the software. Nouns are also shared among the different users and programmers of the system. It is important that different parts of the system use some agreed upon fundamental Nouns in order for the system to work together as a single unit.

In terms of a development process, the fundamental Nouns should be defined and changed though a well defined group process. Once they are defined, they will be used and their functionality relied upon by many different people and systems. For this reason, their design and evolution should be conservative. They should not change too frequently, and the changes should be well thought out.

Verbs are shared object classes that perform some action on a set of Nouns. Specifically, Verbs are java classes that implement the Verb interface and follow a few rules. Examples of Verbs are: MakeGalaxyList, PaintFrame, FitLightCurve, CalculateDustCoef, and RunMission

Verbs allow many different people to use Nouns to perform actions without needing access to the Noun source code. In a traditional OO approach, the actions would become methods of some class (Noun) or another. There are two ways to accomplish this. In the first, there is a central owner of each Noun and if someone else wants additional functionality added to that class they need to ask the Noun owner to add it. If the functionality of they Noun is well understood and stable this can work well, but if different people with different expertise need to play around to figure out the correct this functionality, this method breaks down. The second method allows any one to add functionality to a Noun. This can lead to classes that lack consistent design and become more and more fragile as different programmers add unrelated code. It also can lead to code branching and version dependencies.

Verbs separate out the non core functionality from the definition of the Noun. Anyone can add an action that acts on any set of Nouns without affecting the stability of the system as a whole. This system works well when you have a distributed development model with different groups trying to use the same data to solve different sets of problems or different parts of a large problem.

To summarize, Nouns should be defined to contain the data that defines an entity that will be used by more than one part of the system or that needs to be saved. The only methods of the Noun that should be added are those that are of broad use, are tightly related to the definition of the Noun, and are stable in their implementation. Verbs should be used for all other functionality. For example, getMagnitude() might be a method of the Noun Galaxy, but MakeGalaxyList should be an Verb.

4.0 Noun Class

4.1 Notes

4.1.1 To create a user defined Noun extend the class Noun.
4.1.2 All Nouns need to have a default constructor in order to support some Collections such as NounTables.
4.1.3 All Nouns should implement a correct toString method
4.1.4 In general, developers should follow all rules outlined in Efficient Java

4.2 ClassID

Right now there is no system to create the ClassIDs. There does need to be a mechanism to insure the ClassIDs are unique. Probably a simple semi-automated system will suffice.

4.3 Noun Class

```
class Noun {

    private Object owner;           // null if the object is not a compositional
                                    // object, else points to the object that
                                    // owns this object.  Used for safety checks
                                    // only--to make sure a compositional object
                                    // is owned by one and only one object.

    /** These are not in version 1.0, but possibly in later versions **/
//  private static OutputAdaper oAdapter = null;    // set to use custom adapter
//  private static InputAdapter iAdapter = null;    // set to use custom adapter

    private boolean     dirty;              // (Possible Implementation Detail!)

    private SNOID       snoid;              // snap object ID -- invariant
    private String      iName;              // instance name  -- user modifiable


    /**
     * These next three methods are Noun Info methods.  They MUST be redefined
     * for each subclass!
     */
    public static String getDescription();
    public static String getVersion();
    public static short  getClassID();

//  public void     makeDirty();

    public SNOID    getSNOID();                 // get objects SNOID
    public string   getIName();                 // get objects current IName
    public void     setIName(string newIName);  // IName is user set name


    public void dispose();          // clean up object if needed.  NOP in Noun.
}
```

4.4 A Simple Example

```
class Supernova extends Noun {
```

```
    private int    Metalicity;
    private double Temperature;
    final   Link   Galaxy       = new Link(Galaxy);

    static {
        /**  Here's where we add the optional DDL code. This will be discussed below. **/
    }

    Supernova(string theName, int aM, double aT, Galaxy aGal) {
        setIName(theName);
        Metalicity = aM;  Temperature = aT;
        Galaxy.set(aGal);
    }

    public static String getDescription()
    {
        return "Supernova Model";
    }

    public static String getVersion()
    {
        return "2.03 -- 1 August 2003";
    }

    public int    getMetalicity()   {  return Metalicity;  }
    public void   setMetalicity(int aMetalicity)  { Metalicity = aMetalicity; }
    public double getTempreature()  {  return Temperature;  }
    public void   setTemp(int aTemp){  Temperature = aTemp; }
    public Galaxy getGalaxy()       {  return Galaxy.get(); }
    public void   setGalaxy(Galaxy aG)  { Galaxy.set(aG);   }
    public void   setGalaxy(SNOID anOID){ Galaxy.set(anOID);}
}
```

4.5 Schema Evolution

For a discussion of how Nouns are involved with Schema Evolutions, see the section on schema evolution below.

5.0 Verbs

An Verb is a class (Component) that does something. It takes a specific set of objects as input, performs an action, and returns a set of objects. Verbs can be as fine grained or course grained as you like. For example, an action could take a calibration and a flux and return a magnitude. Or, an action could simulate a mission, taking as input a mission plan and returning as output cosmological parameters.

The current implementation of Verb uses standard java dynamic binding and name space services, so all actions are available to implementers at all times.

Uses for Verbs :

- The primary function of an Verb is to provide functionality in a controlled manner without modifying the source code of a Noun.
- Verbs enable easy scripted or interactive operation.

- They can be used as Components.
  - To black-box different aspects of the simulation or pipeline. For example, an Verb could be CLEAN, that uses other actions to produce a cleaned image.
  - Computationally expensive Components that can allow us to distribute parts of the system at a later time. For example, FITLIGHTCURVE could run at LBL and CLEAN could run at FNAL.

5.1 Notes

5.1.1 Design Considerations

From a design perspective, highly granular algorithms that do one things leads to maximum flexibility. These base algorithms can then be used to build up more complicated Verbs. Sometimes, however, in cases where the base actions are not going to be used repeatedly, or where performance can be optimized, it makes sense to create large monolithic do-all Verbs.

5.1.2 Saving Verb Configuration Parameters

Verbs can extend “Noun” also if you want to be able to save out the state of the configuration parameters.

5.1.3 Verbs calling other Verbs

Verbs can call other actions without restrictions.

5.2 Parameters Format

Input and output parameters will sent in via NounContainers. Currently there are three NounContainer : NounTable and NounList and NounMap.

5.3 Interface

~~5.3.1~~ ~~Verbs should use a Factory Method instead of a constructor in order to facilitate moving to a distributed framework at some point.~~

```
~~private Verb();          // no default constructor~~
~~public Verb newVerb();~~
```

5.3.2 Singleton

If appropriate, it is entirely exactable to make an Verb a singleton. It is not clear yet, but this may turn out to be the preferred and most efficient design.

5.3.3 Object Identification Methods

```
String getName : Name of algorithm.  Invariant.
String getVersion : Version of algorithm.  Invariant.
```

5.3.4 Execution Methods

```
NounCollection run() : run the algorithm
NounCollection run(NounCollection input) : set the params
      and run.
```

5.3.5 Execution Helper Methods

Optional but suggested methods to allow an algorithm to be called without using a Container for parameters can be implemented. These helper methods may be called `run` and they take a normal java coma delimited set of parameters. For example :

```
FitLightCurve:
run(LightCurve lc);
```

```
MeasureSNFlux:
run(Supernova sn, LightCurve lc);

Alternative (document)
Map getInputMap(); // map with fields filled out
```

6.0 DistributedVerb

At some point, it might useful to distribute the Verbs in a cluster.

7.0 Relationships

There are two kinds of relationships between Nouns supported by the system: *Compositions* and *Links*. Both kinds of relationships are one directional, with one Noun ***owning*** the relationship and the other Noun ***participating*** in the relationship. The relationship itself is an object and is named. One ***traverses*** a relationship to get from the owner to the participant.

Here's an example: There is a Noun called Dog and a Noun called City. Each Dog lives in one city. So, that would be represented by a relationship called LivesIn and the Dog Noun would own the relationships. Specifically, Fluffy owns a relationship called LivesIn that points to Berkeley. Since Dog owns the relationship, if you had access to the Dog instance Fluffy, you would be able to traverse the LivesIn relationship to get the City instance Berkeley. Since the relationship is one way, if you had access to Berkeley, there would be nothing to traverse to get back to Fluffy.

A compositional relationship is when one Noun uses another Noun as part of its definition. In OO terminology this is called a has-a relationship. A particular instance can participated in exactly one compositional relationship. For example, a Customer has-an Address. Since this is a compositional relationship the Address becomes part of the definition of Customer. As such, each Address in a Customer is a unique object. Even if two customers live in the same location, they each have a different Address instance. So, if one Customer modifies her Address, the other Customer's Address is not affected. This is enforced by the system.

The other kind of relationship is a Link. In OO terms this is simply called being related. A husband is, in America, related to one wife. Still, one object owns the relationship. We could make it so that the Husband Noun owns the MarriedTo relationship. Then if you had the Tom instance, you could traverse the MarriedTo relationship to get to Tina. An instance can participate in many Links. For example, Tina can participate in the Wife Link of Tom and the CFO Link of the Company Instance WidgetMaker.

When an object is saved, all objects related to it via a Composition will also be saved. In our example, when a Customer is saved, the customer's Address is also saved. When an object with a Link is saved, the object that participates in the Link is <u>not</u> saved. So, when Husband Tom is saved, Wife Tina will not be saved.
While Compositions are by nature not polymorphic, Links are. If the Person class has a Relationship called Friend where the participant was an instance of the Person class, the participant could be of Noun Person, or Husband, Wife, Boss, or any subclass of Person. Links do not have to be defined between two Nouns. A Link can also be defined between a Noun and an interface. However, if during runtime a Link is made to an object that is not derived from Noun then many framework services will throw an exception. In Practice, through good design this can be completely prevented at compile time.

If you want the link to be bi-directional, then you must put relationships in each objects. So, the Noun Wife owns a MarriedTo relationship that points to a Husband, and the Husband Noun owns a MarriedTo relationship that points to a Wife. There is no referential integrity by the system—it is up to the programmer to insure that a Husband, Wife pair always point to each other.

If something is Linked to or Composed to more than one object, then the participating Noun should be a container. So a Dog could be related to a NounList of Hairs.

7.1 Relationship Helper Classes

```
//  used by Relationships and probably some collection class functions
class LazyPointer {
           SNOID    oid;      // object pointer
           Object   obj;      // object

public:
           object get();
           void   set(object O);
}


class Relationship {
           private LazyPointer      ptr;

           /** The default for Composition is false and for Link it is True **/
           void lazyLoad(boolean shouldLLoad);
           ...
}
```

7.2 Link Class

```
class Link extends Relationship {
           /**
            * Create a link to an object of Noun linkNoun.  Calling set with an object
            * of any other type is an error and exception will be thrown.  This will
            * probably be enforced using an assert, so the checking can be turned off
            * for performance.
            *
            * Initially the Link points to no objects, so get() will return null and
            * getSNOID() will return SNOID.NULL.
            */
           public Link(Noun linkNoun);

           public void set(Noun  newLink);
           public void set(SNOID newLink);

           /**
            * Relationships are by default LazyLoaded.  If the related object has not
            * been loaded, calling get can cause an IO operation.
            */
           public Noun  get();

           /**
            * Calling getSNOID will never cause an IO operation.
            */
```

```
            public SNOID getSNOID();  // will *not* cause the object to be loaded!
}
```

7.3 Composition Class

```
class Composition extends Relationship {
            /**
             * Create a new composition of Noun compNoun.  The compositional object
             * will automatically be created and will never change, although the
             * attributes of the compositional object can change.
             */
            public Composition(Noun compNoun);

            public Noun  get();
            public SNOID getSNOID();
}
```

7.4 Simple Example of a Link and Composition

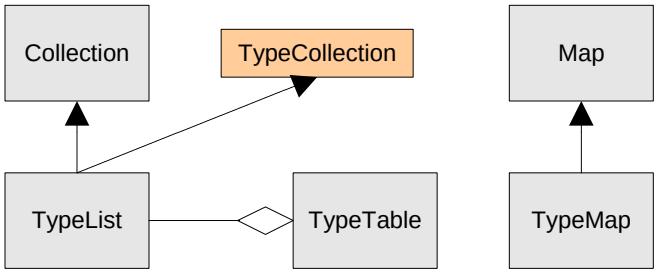
```
class Shape extends Noun {
    ...  Shape Stuff Goes Here ...
}

class Galaxy extends Noun {

    /**
     * Here we create a Composition Relationship.  Galaxy is composed of a
     * shape object.    Composition creates the shape object automatically!
     */
    public final Composition theShape = new Composition(Shape)
    ...
}

    /**
     * Here we create a Link Relationship.  Supernova is related to a Galaxy.
     */
class Supernova extends Noun {
    public final Link inGalaxy = new Link(Galaxy);
    ...
}


boolean compareShapes(Galaxy g1, Galaxy g2)
{
    Shape s1 = g1.theShape.get();
    Shape s2 = g2.theShape.get();

    return s1.equals(s2);
}
```

8.0 Collections

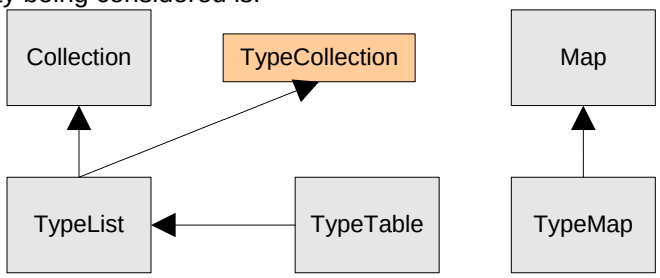
8.1 Notes

8.1.1 Collection Classes

The following collection classes are provided by the framework: NounTable, NounList, (NounLazyList), NounMap. Below is a possible class hierarchy.



The other possibility being considered is:



8.1.2 If at all possible, the collections will implement the Java standard Collections interfaces. If it isn't possible, then NounCollection will be used to model the interface as closely as possible.

8.1.3 To understand the API that will be used, look at the standard Java collection classes: ArrayList and Map.

8.1.4 Collections sometimes hold objects, sometimes LazyPointers, but the user always interacts with collections the same way.

8.1.5 There will be added methods to dispose the Collections: Dispose (Shallow) and DisposeAll (Deep)

8.1.6 Collections can be used to implement *Events* or *Catalogs*. Catalogs are complicated enough and can be optimized enough that it is probably better to create a Noun that presents the Catalog functionality.

8.1.7 Collections can be persisted either deeply or shallowly using the Directory.

9.0 NounTable

9.1 Notes

9.1.1 General Notes

SNAPTable is a NounCollection class that behaves like a table. It is unclear at this point if NounTable will extend NounList or just allow users to go back and forth between the two. Since java doesn't perform automatic conversions this is an important decision.

9.1.2 Performance

Because tables parse the column names at runtime they are inherently less efficient than dealing with objects directly using something like a NounList. When using the member selection operator (dot operator) the compiler does the hard work and compiler generated code can often be optimized to a short inline function. NounTable is also implemented using reflection and this might exacerbate the performance penalty. For data manipulation not in tight loops this should not be an issue.

9.2 Class Definition

```
/**
 * It is unclear if NounTables will work with compositional objects in the
 * early versions.
 */
class NounTable extends NounList {
    /**
     * Create a new NounTable for the given Noun.
     *
     * Example:
     *      NounTable SNTable = new NounTable(Supernova.class);
     *
     *      SNTable.setDouble("Metalicity", 25.5);
     *      SNTable.setLink("Galaxy", gal);
     *
     *      int t = SNTable.getInt("Temperature");
     */
    public NounTable(Noun defineTheColumns);

    /**
     * Create a new NounTable with the objects from the given collection.
     * The objects are NOT copied, so this creates an ALIAS collection!
     *
     * Example:
     * public NounCollection fitLC(NounCollection LC_Col) {
     *      NounTable LCTable = new NounTable(LC_Col);
     *          ...
     *      return LCTable;
     * }
     */
    public NounTable(NounCollection aCollection);

    /**
     * Take a NounTable and return a NounList.
     */
    public NounCollection getNounList();

    //  Add a new row default to the table
    public void addRow();

    //  set the cursor to a row;
    public void setWorkingRow(int i)

    //  methods to get and set the columns

    public int getInt(string colName);
    public void setInt(string ColName, int value);

    public double getDouble(String colName);
    public void setDouble(string colName, double value);

    public Noun getLink(string colName);
    public void setLink(string colName, Noun linkObject);
}
```

10.0 Directory

The Directory is the lowest level user interface to the persistent store. For the simulation project users will use DataSets which in turn will be implemented using the Directory.

10.1 Notes

10.1.1 Each implementation should (probably) be a singleton.

10.2 Interface without comments

```
interface Directory
{
    Directory getDirectory();
    ftoken openDatabase(string name, unsigned flag);
    closeDatabase(ftoken);
    rtoken openRecordList(ftoken, string name, unsigned flag);

        /*********************/
        /** Record List API **/
        /*********************/

    void AdvanceRecord(rtoken);
    Noun getFromRecord(rtoken, class);
    Collection getAllFromRecord(rtoken, class);
    void writeToRecord(rtoken, Noun);
    void writeToRecord(rtoken, Collection);

        /*********************/   /**************/
        /** Repository API **/   /** Phase II **/
        /*********************/   /**************/
    collection select(ftoken, class, Selector);
    collection select(ftoken, class, string OQL);
    save(ftoken, Noun);
    save(ftoken, SNCollection);
    save(ftoken, SNMap);
    save(ftoken, SNCollection, boolean deep);
    save(ftoken, SNMap, boolean deep);
}
```

10.3 Interface with comments

```
/**
 * All Gets are polymorphic.  That means if you get AstronomicalObject you will
 * get both Supernovas and galaxies.
 */
interface Directory
{
    /**
     * Get the current Directory.  Directories will probably be implimented as a
     * Singleton but users should not rely on this.
     */
    Directory getDirectory();

    /**
     * Open up a database.  A database contains objects and "RecordLists".
```

```
     *
     * flag can be some combinaion of :
     *   R : Read
     *   W : Write
     *   C : Create
     *
     * Other names that could used in the final spec:
     *   openFile(  )
     *   openNamespace(  )
     */
    ftoken openDatabase(string name, unsigned flag);


    /**
     * Open a particular record list for Input or output.
     * A RecordList is a named collection of records.  Each
     * record contains one or more related objects.
     */
    rtoken openRecordList(ftoken, string name, unsigned flag);

        /*********************/
        /** Record List API **/
        /*********************/

    /** Advance to next Record **/
    void AdvanceRecord(rtoken);

    /**
     * Get an object of a particular type from the current record.  If there is
     * more than one object of the specified type then it is undefined which
     * object will be returned.
     *
     * Example:
     *
     * supernova sn = getFromRecord(rt, Supernova);
     */
    Noun getFromRecord(rtoken, class);

    /**
     * Get all objects from a record of a particular type.
     *
     * Example:
     *
     * SNTable snt = getAllFromRecord(rt, Supernova);
     * SNList  snl = getAllFromRecord(rt, Supernova);
     */
    Collection getAllFromRecord(rtoken, class);

    /** Load an instance of Noun that has the specified iName **/
    Noun get(ftoken, Class Noun, string iName);

    /** Load an instance of Noun that has the specified SNOID **/
    Noun get(ftoken, Class Noun, SNOID);

    /**
     * Write an object to the current record.  Objects in the same record are
```

```
     * assumed to be related in some way.  All objects written using this API
     * are "new" objects.  This has implications for relationships.  See
     * the Noun class for a more detailed discussion of the implications.
     */
    void writeToRecord(rtoken, Noun);

    /**  This is a deep save **/
    void writeToRecord(rtoken, Collection);

        /********************/   /**************/
        /** Repository API **/   /** Phase II **/
        /********************/   /**************/

    /**
     * return a collection of objects that return true for the given Selector.
     *
     * Example:
     *
     * Selector test = new Selector() {
     *                           test(Noun t) {
     *                               Supernova s = (Cosmology) t;
     *                               return (s.Noun == 'IB');
     *                           }  }
     *
     * SNTable snt = select(ft, Supernova, test);
     */
    collection select(ftoken, class, Selector);

    /** For performance... later **/
    collection select(ftoken, class, string OQL);

    save(ftoken, Noun);
    save(ftoken, SNCollection);
    save(ftoken, SNMap);
    save(ftoken, SNCollection, boolean deep);
    save(ftoken, SNMap, boolean deep);
}

interface Selector {
    boolean test(Noun t);
}
```

11.0 Meta Data Repository (MDR)

The Meta Data Repository (MDR) stores information about all the Nouns currently in use. It is currently only used by the Input / Output subsystems, but could be used by other systems, such as visualization, if desired. The MDR has other methods used internally to access the data, but those methods are not yet documented because they are not yet stable.

Currently, the only way to add meta data is through the programmatic interface. In the future, it would be possible to write an XML loader or to use some other Data Definition Language (DDL) to populate the MDR.

11.1 Important Note

It is very important that you define meta data only for data members defined in the current class and not in inherited bases classes! The MDR will attempt to detect errors of this type, but ierror detection may not be guaranteed. In case, defining meta data for inherited data members will produce undefined results.

11.2 THINGS TO ADD

The MDR could (and should) be able to get all the information on the data members and using the defaults create all the necessary meta data fields. We could require at least the addNewClass call, or could just add it the first time it is used.

11.3 Temporary place for Implementation Notes

All of the Get methods should cache a certain number of requests!

11.4 Interface

```
class MDR (Meta Data Repository)  {
           /**
            * Returns a List of classes that are the given Noun.  For example,
            * if Star and Dust were derived from AstronomicalObject, then calling
            * getIsAList(AstronomicalObject.class) would return a list of all three
            * classes.
            *
            * The List contains MClass objects.  MClass objects store all the
            * meta data about a given class including the name of an instance of the
            * class object.  All MClass objects are read only!
            */
           public List
           getIsAList(Class aClass);

           /**
            * Returns a List of all attributes of a given Noun.  This includes both
            * the attributes defined in class and all inherited attributes.
            *
            * The List contains MAttribute objects.  The MAttribute contains all the
            * meta data about a given attribute including the name and type.
            */
           public List
           getAttributeList(Class aClass);

           /**
            * Returns a List of all the relationships of a given Noun.  This inludes
            * both the relationships defined in the class and all inherited
            * relationships.
            *
            * The List contains MRelationship objects.  The MRelationship contains all
            * the meta data about a given relationship including the name and type.
            */
           public List
           getRelationshipList(Class aClass);

    /**
     * AddNoun adds a new type to the Meta Data Repository.  It uses
     * Reflection to figure out what the new type extends and what interfaces
     * it impliments.  Internally it creates an object model and list of what
     * classes impliment which interfaces.
     *
     *      [Noun]        [Interface1]        [Interface2]
```

```
     *       / \             /  \                 /  \
     *    [SN] [Gal]       [T1] [T2]           [T1] [T3]
     */
    addNoun(Class newNoun);
    setSQLTableName(Class aNoun, String theTableName);

    /**
     * AddData adds a new data memeber to the definition of a Noun in the MDR.
     * The MDR class defines a set of constants to represent the data types
     * that the repository knows how to represent.  Examples include
     * long, double, complex
     */
    addData(Class theNoun, string dataName, int DataNoun);

    /**
     * There are a set of methods that set attributes for the data members.
     * Each attribute has a well defined and stable default, so that for
     * many data members the user doesn't have to set any attributes.
     * The currently proposed defaults are :
     *      Null      : false (can not be null)
     *      Missing   : false (can not be missing)
     *      Invariant : false (can be changed)
     *      DefFormat : Noun dependent -- will be documented
     *      SQLNoun   : Noun dependent -- will be documented
     *      SQLName   : null
     *
     * Declaring a data member to be Invariant means that once an object is saved, the
     * system will not allow the object to be updated if this data member has changed.
     */
    setDataNull(Class theNoun, string dataName, bool canBeNull);
    setDataMissing(Class theNoun, string dataName, bool canBeMissing);
    setDataInvariant(class theNoun, string dataName, bool isInvariant);
    setDataDefaultFormat(Class theNoun, string dataName, string defaultFormat);
    setDataSQLNoun(Class theNoun, string dataName, string SQLNoun);
    setDataSQLName(class theNoun, string dataName string SQLName); // try not to use this

    /**
     * The next two methods are for creating meta data for Noun relationships.
     */
    addLink(Class theNoun, string linkName);
    addComposition(Class theNoun, string compositionName);
}
```

11.5 Simple Example

```
class Supernova extends Noun, implements SupernovaModel {

    /** Define Data Memebers **/

    private int    temp;
    private double metalicity;

    private final Link galaxy = new Link(Galaxy);

    /** Add Meta Data to Repository -- two examples, choose one **/
```

```
    /* This is what a minimal DDL block could look like */
    static {
        MDR.addNoun(Supernova.class);
        MDR.addData(Supernova.class, "temp", MDR.int);
        MDR.addData(Supernova.class, "metalicity", MDR.double)
        MDR.addLink(Supernova.class, "galaxy");
    }

    /* This is what a complex DDL block could look like */
    static {
        Class snc = Supernova.class;

        MDR.addNoun(snc);

        MDR.addData(snc, "temp", MDR.int);
        MDR.setDataNull(snc, "temp", true);
        MDR.setDataMissing(snc, "temp", true);
        MDR.setDataDefaultFormat(snc, "temp", "0000");
        MDR.serDataSQLNoun(snc, "temp", "I8");

        MDR.addData(snc, "metalicity", MDR.double);
        MDR.setDataNull(snc, "metalicity", true);
        MDR.setDataMissing(snc, "metalicity", true);
        MDR.setDataDefaultFormat(snc, "metalicity", "0XXXX.XXX");
        MDR.serDataSQLNoun(snc, "metalicity", "F8");

        MDR.addLink(snc, "galaxy);
    }

    /** Then the rest of the class is just as would be in java **/

    public
    Supernova(int aTemp, double aT)
    {
        temp = aTemp;   metalicity = aT;
    }

    public
    int getTemp() {  return temp; }

    public
    void setTemp(int aTemp)  {  temp = aTemp;  }

    public
    double calculateMag(double lambda)
    {
        ...
    }
}
```

12.0 DynamicNouns

A DynamicNoun is a standard way to create a type that can have data fields added at runtime and still be persisted. It is meant to be used during development and hand analysis. This will be documented in the future. (Currently it is in GK's notes from Marseille)

13.0 SNAPException

There will be an exception defined for framework errors

13.1 SNAPException

14.0 SNAP OID – Object ID ("SNOID")

14.1 Notes

14.1.1 Definition

Every instance of a Noun has a unique SNOID. The SNAP Object ID (SNOID) is the underlying invariant name of each object. In some systems this functionality is provide by a URI. The SNOID is the *primary key* and most fundamental name of any object. Objects may be loaded from the SNOID.

14.1.2 Opaque

The SNOID is an opaque object. From a user perspective, it is a string of characters that means nothing.

14.1.3 There is a class SNOID

As stated, the class should be treated as opaque and SNOIDs should only be compared using the .equals method. However, in the current implementation the SNOID object contains the SNOID itself and also a *lock count*. The lock count is stored in the database and used for optimistic locking.

14.1.4 Relationships

Relationships between object instances are actually relationships between SNOIDs. The value of any data field can change, and the relationship will still point to the *same* object because same is defined as having the same SNOID.

As an example, imagine that Husband ["George"] is Related to Wife ["Sally"]. That instance of Husband has some SNOID, say 1. Wife has some SNOID, say 2. The Husband with SNOID 1, will always point the Wife with SNOID 2—unless the Relationship is explicitly changed. So, if in the code the Wife instance with SNOID 2 is changed to ["Mary"] (*wife.setName("Mary")*), then ["George"] will point to ["Mary"] because nothing changed the SNOID of that instance of Wife.

This behavior is exactly the same as what would be expected in Java if one object pointed to another object directly and this is **usually** the desired behavior, but it is crucial to keep in mind as objects are modified. For example, if an Exposure object points to a Calibration objects and then that Calibration object is modified, the Exposure will point to the modified Calibration object. In this case, the behavior does not produce the desired result. The correct behavior would be to create a new Calibration object and not an existing one.

In order to prevent this situation, always create a new object if a modification would result in the object becoming a logically distinct entity. For example, even if a Calibration is only slightly changed, any modification to the Calibration object results in a new Calibration.

[The following paragraph will be deleted, I'm leaving it now so that the people who where at the last workshop will understand why this functionality is not in this draft even though we said it would be.
[In the cases where you want to start with one object and then modify it so it becomes another object, there is a method of type called **xxxxxxx** that takes the object and gives it a new SNOID so that that instance is not associated to any other object. **This method can't exist because there would be no way to sever the in memory relationships. It is up to the programmer to create a new object.**]]

Note : On even further thinking we should be able to provide this functionality if we go about it a little differently. Instead of having a method that takes a Noun and makes it so that that Noun is not attached to anything (gives it a new SNOID), we can just have a method that returns a new Java reference. I can think of two ways to do this right now : either require a copy constructor for each Noun, or add a generic method to OIL. I think this works?

14.2 Possible Format

Class ID : Version ID : Sequence Number
(63 bits)

Where Sequence number is defined like this: Milliseconds since Jan 1, 2000 : XXX

Where XXX is the MSecondCount. MSecondCount is incremented each time a new SNOID is requested in the same millisecond. This should allow for 4096 objects allocations per millisecond.

So, the getSNOID algorithm looks like this:

```
// Ignoring the Class ID : Version parts -- SNOIDS are hexidecimal
SNOID getSNOID()
{
    static short secondCount = 0;
    static long  lastSNOID   = 0;

    long newSNOID = getMSecs();

    newSNOID << enough;             // move over to make room for SecondCount

    if (lastSNOID && MASK == newSNOID) { // mask out any existing SecondCount
        newSNOID |= ++secondCount;
        if (secondCound >= max) {        // this check is hardly worth it
            pause(.1 second);
        }
    }
    else {
        secondCount = 0;
    }

    return newSNOID;
}
```

15.0 Block Serial Number

To aid in reproducibility and data consistency we may implement a write only data store such that each write creates a new data block with a unique serial number. The current plan is to use the SNOID as the Block Serial Number.

16.0 Service: Logging

Provide unified logging and debugging information.

17.0 Service : DataIO Interface

The DataIO service is the low level service that will implement persistent storage. There might be one version for flat files and one version for databases. Users will interact with the directory and not with the DataIO Service.

18.0 Utility Nouns

18.1 RuntimeValues

A place to store system wide run time information. It will probably end up being a Map.

```
Public class RuntimeValues {
    Public final string LOGGER = “LOGGER”;
    public static Map get();
}
```

18.2 PhysicsConstants

A place for physics constants. Probably should be a SNAPNoun so it can be versioned and persisted???

19.0 Utility Verbs or Class Libraries

This functionality will be provided by the system. It will either be simple java classes, or Verb Components depending on what makes more sense.

19.1 Visualizations
19.2 Histograms
19.3 Data Browsers
19.4 Matrix Arithmetic (Colt)
19.5 Vector Arithmetic

Pull a column or data field out of a collection. Operate on it. Put it back. (from FNAL)
Note: Could be implemented as an action that takes out + an action that puts back.

19.6 FITS I/O Library (Goddard?)
19.7 Complex Numbers

20.0 Batch Format and Prompt Language

Jython will be used as the batch language and interactive prompt. Details and examples to follow.

21.0 Interactive GUI

Possibly there will be an interactive GUI.

HISTORY AND REPRODUCIBILITY

22.0 Record Based File Format (*TENTATIVE and Work in Progress*)

Until a full database repository is implemented, file system based storage will be used.

23.0 SQL Schema

Random Notes

Supernova Table

Block ID	SNOID	M	T	X	Y

There will need to be a service to create the schema.

24.0 SQL Access Layer

Using the OQL services in the Directory, users will be able to interact with the data store in a SQL like manner with SQL level efficiency.

The examples in this section are all out of date. They are being left in, so that they can be updated as time permits !!!

25.0 Section (Examples)

Relationship Example (SN has-a LC)
Batch Example
Interactive Example
Schema Evolution Example

26.0 Some Code Examples

26.1.1 Batch File (Could also be typed in interactively or done via a main)

Algorithm	What Happens
CreateSupernova "Test" T=200 Metalicity=25	A new supernova instance is created with the initial parameters specified. The object is named "Test" and it is registered with the directory.
CreateLightCurve "Test-LC"	Create an empty instance of LC and register it with the Directory.
CreateLightCurve "Test-LC-G"	Create an empty instance of LC for the green filter measurements.
Loop Day = 5 Times Begin	This won't be the batch file construct, but there will be some way to specify looping.
TakeLightMeasurement "Test" "Test-LC" Day	Calculate the flux (or mag) for the given day. Add the Entry (Day, Flux) to Test-LC
TakeGreenLightMeasurement "Test" "Test-LC-G" Day	Same as above, but use a procedure to apply the Green Filter.
END	End the loop
Save "Test-LC"	Write the LC to the default output stream
Save "Test-LC-G"	Write the green LC.

27.0 A Simple Simulation Example (1)

This example is of a simulation run that takes a "True Universe" creates a bunch of Supernovas, then measures the light curves, then fits the light curves, then calculates a "Measured Universe". Right now it is missing **most** of the pieces—I'll fill them in later as I get time. But, hopefully, even in this form it conveys some useful information.

27.1 Simulation Diagram

28.0 Example : A Relationship

29.0 Example of reading in Measured LCs, Fitting them, and then writing them out.

30.0 Example: MakeSNEvents

In this example, we take a universe and SNDistribution and create a SNEventList. We assume we are given the Universe instance and the SNDistribution instance.
To do : Create SNEventList (algorithm), and SNEventList (Model)

APPENDICES

31.0 Appendix I : Coding Conventions

These are just suggested…

31.1 Gosling invented the language, use his coding style.
31.2 Follow Effective Java unless you understand exactly why in your case it isn't correct. To prove understanding you must document the
31.3 Purchase and Read : Core Java Volumes I and II & Effective Java

32.0 Appendix II : Requirements for Mission or Science Pipeline

32.1 Work Flow system + Monitoring + Restart Ability
32.2 Software rollout steps
32.3 Version upgrade steps
32.4 Image and File Protection

When we're running with real data we need a mechanism that protects data at the filesystem or database level. Examples of data that will need to be protected using this mechanism are : raw telescope images, published data, etc.

33.0 Appendix III : Random Notes on Relationships

34.0 Appendix III – Thoughts on schema Evolution

Take 2:

34.1 To add a data member

Add it. If you want old data to still load, mark the new data member as ok to be missing. Then in code using the new object definition, you must be able to handle the case where the new data member is null or the default value.

34.2 To delete a data member

Just delete it. It will be ignored on load w/o error. If the schema is regenerated the deleted data member will not be included.

35.0 Appendix V : Random Notes

35.1 Current Status

Some factors in the decision are how well each system allows for :

- Handling Relationships (SN → Gal) (Optics → Optics)
- Noun Safety
- Schema Evolution
- Ease of use and understanding
- Maintaining and understandable and robust system as complexity and size grow
- Others

35.2 Assumptions

35.2.1 Sometimes sequential mode (one record at a time)

Will not have lots of records in memory at any one time, but lots of record I/O.

35.2.2 Sometimes will be parallel mode (lots of records at once)

Will need to store and access lots of records at a time. Indexing/Access is crucial as is memory management.

35.3 Questions

35.3.1 How will objects be linked efficiently.

For example : LC → SN → GAL