GORP: An Object-Oriented Design for Genomic Objects, Relationships and Processes

Robert Mecklenburg
Tony Di Sera
Peter Cartwright

UUCS-96-007

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112 USA

June 14, 1996

Abstract

The Eccles Institute for Human Genetics (EIHG) has developed a genomic database based on a novel level of abstraction. Objects, relationships, and processes are explicitly represented in an object model. This model has been implemented in a traditional relational database management system. Translating this database model into an object-oriented programming language has proven to be a challenging exercise. This paper describes a C++ object-oriented design which faithfully implements a persistent object model of the original abstractions. In addition, a new level of meta-information is provided which enhances an application's ability to adapt to changes in the database schema. This is a description of work in progress and includes a survey of relationships and processes in the current EIHG databases.

1. Motivation and Approach

1.1 Historical Context

The Genome Software Development Group (called *The Group*) has been tasked with writing a software system to:

- record human genome sequence;
- record abstract higher-level genetic structures (e.g., genes, introns, exons);
- help manage the daily problems of large-scale sequencing processes;
- help manage people and resources related to sequencing.

The approach taken to achieve this software system is based on an object-oriented model (called *The Model*). The Model is based on five concepts:

1. Objects — These are "nouns" or things in the world (typically physical things).
2. Relationships — Objects participate in a set of relationships with other objects. Relationships can be simple like "contains" or can be more complex like "is derived from". Simple relationships have no additional data except the relationships and are called *thin relationships*. Complex relationships have associated data and are called *thick relationships*.
3. Processes — Processes are events in the lab or world that operate on Objects and Relationships and produces new Objects and Relationships. Processes can be software-oriented or physical behavior-oriented.
4. Protocols — Processes are performed according to protocols. Protocols can be altered over time and can be altered on a per process-basis.
5. Environment — When a process is executed, that execution takes place within an environment which includes the materials and equipment used, times, temperatures, etc.

Four of these concepts are diagrammed in Figure 1. The effort of the Group has been divided between writing applications to address tasks in the lab and implementing the Model as a database to record lab data. The realities of software development have forced the Group to use traditional relational database software to implement this OO model.

One component that has not been addressed in an orderly manner is the realization of the Model in an object-oriented programming language (OOPL). Rather, the database support software has been somewhat enhanced to export some of these concepts and applications have added support for these features in an ad hoc manner.

1.2 GORP

The purpose of this work is to revisit the Model and to develop an OOPL specification. This specification should be implementable in any modern object-oriented programming language. The design is guided by several basic goals:

- faithfulness to the Model;
- desire to insulate applications from implementation details;
- use of object-oriented techniques;
- support for incremental implementation and enhancement.

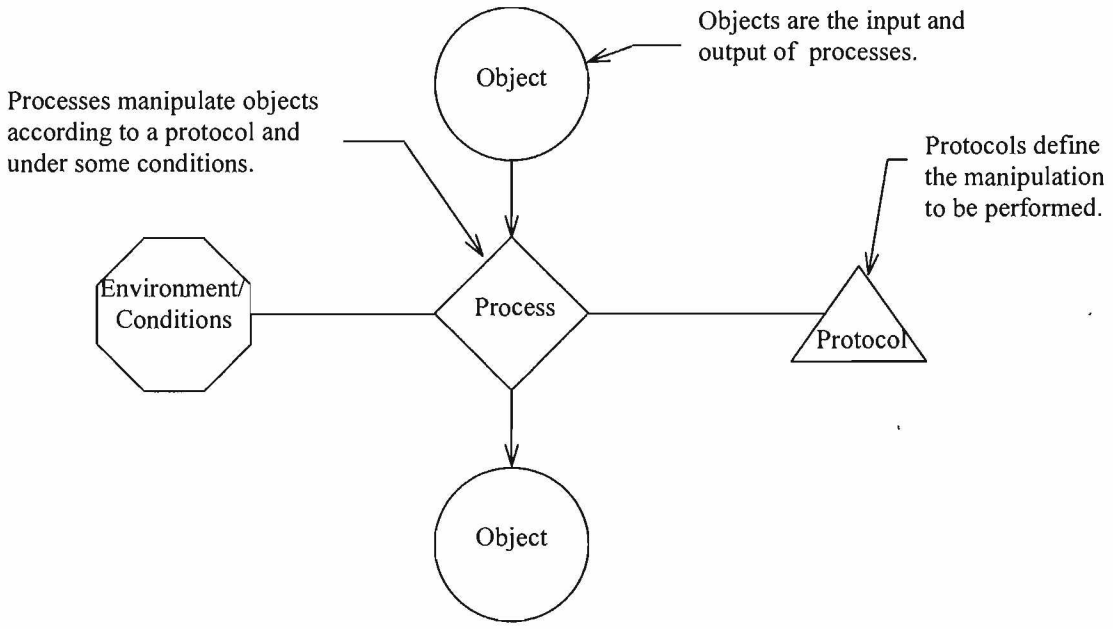


Figure 1: Overview of the Model.

We discuss each point in turn.

The Model is the result of many years of effort and experience. In particular, the novel use of relationships is intended to provide a *schema evolution* mechanism outside the traditional relational model and is essential to the long term success of the project. Any implementation of the Model in an OOPL must address the use of this technique at the programming language level. This work is intended to fill-in a missing portion of the Model rather than to replace it. As such, the techniques developed during the implementation in an RDBMS will most likely be retained and expanded. The OO layer proposed here should mesh cleanly with the integrity constraints in the RDBMS and hopefully add enhanced security at the language level.

Current applications are written with an intimate knowledge of the existence of the underlying database software. Data types and access methods are directly used by application programmers to implement functionality. This leads to confusion about how the Model should be manipulated at the application level. The confusion arises because application programmers view the Model as an RDBMS consisting of aggregates of primitive types similar to C structures, rather than a set of concepts embodying state, behavior and constraints. In addition, this knowledge makes replacing the existing database software with alternative implementations much more difficult. Finally, application programmers would rather not know about the database since, to a large extent, its existence is irrelevant to the application. So by hiding the implementation of an object behind a strict interface, the persistent objects can be structured to allow a variety of implementations. For instance, the basic technique for retrieving an object should be to perform some type of generic *get* function. Such a function can be overloaded to accept constraints, unique ids, etc. This allows the *get* function to perform database retrieval, or manage flat files, or sockets, etc.

The Model diagrammed in Figure 1 is not fundamentally an object-oriented model as viewed by the OOPL community. The "thing" called "Object" is really just intended to distinguish nouns from verbs (e.g., processes) and adjectives (e.g., protocols and environment and conditions). Nevertheless, it is quite reasonable to apply object-oriented methods to the design of the programming language interface and indeed that was the original intent of the use of the word *object* in the Model. Circumstance and lack of experience have lead to the current confusion. In an object-oriented design, the fundamental elements of the problem domain are the focus of the design effort. These objects are defined as rigorously as possible and include state, behavior, constraints, and capabilities. Elaborating these objects and their characteristics is the purpose of this work.

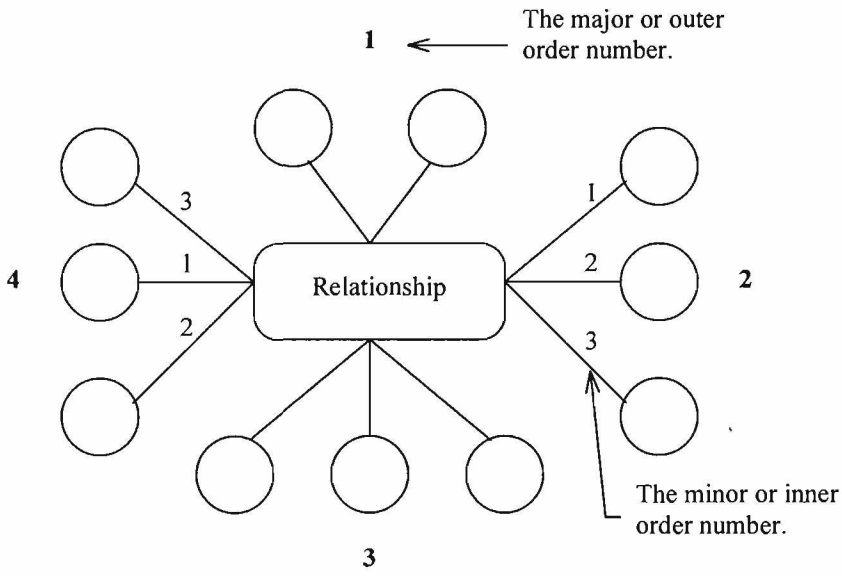


Figure 2: The Relationship Octopus.

Finally, the current system is "live" and in use daily. It is important to allow incremental implementation of functionality in the library and incremental evolution of applications to use the new features. Mostly this means ensuring that existing uses of the persistent object database continue to perform their functions while adapting to the new object structure. An alternative to performing radical changes to the database and all applications simultaneously would likely be unacceptable due to real computing requirements and deadlines in the lab. Another alternative, replacing the existing applications and database is also difficult to justify due to the immense investment in programming effort and the relative success of these applications.

1.2.1 Relationships

Objects in the Model participate in relationships with other objects and relationships. In a traditional relational model these relationships might be implemented as sets of primary and foreign keys in tables defined for each relationship. The Model structures these relationships into a universal many-to-many join table. See Figure 3 for an attempt at explaining this graphically. One advantage of this approach is that new relationships can be created dynamically without the overhead of evolving a database schema. A relationship in the Model has a very general formulation which allows for fairly complex linkages. A member of the relationship is tagged with two numbers called the major and minor order (or outer and inner order, respectively). This is diagrammed in Figure 2.

Relationships in the Model correspond (roughly) to pointers or references in an OOPL. Relationships in the Model differ from pointers in that they can be one-to-one, many-to-one, or many-to-many, and these relationships are often bi-directional. In this implementation, creating a new relationship means adding a data member which references the new set of objects. Again, this corresponds to evolving the schema. Thus the OOPL implementation subverts the intent of the Model.

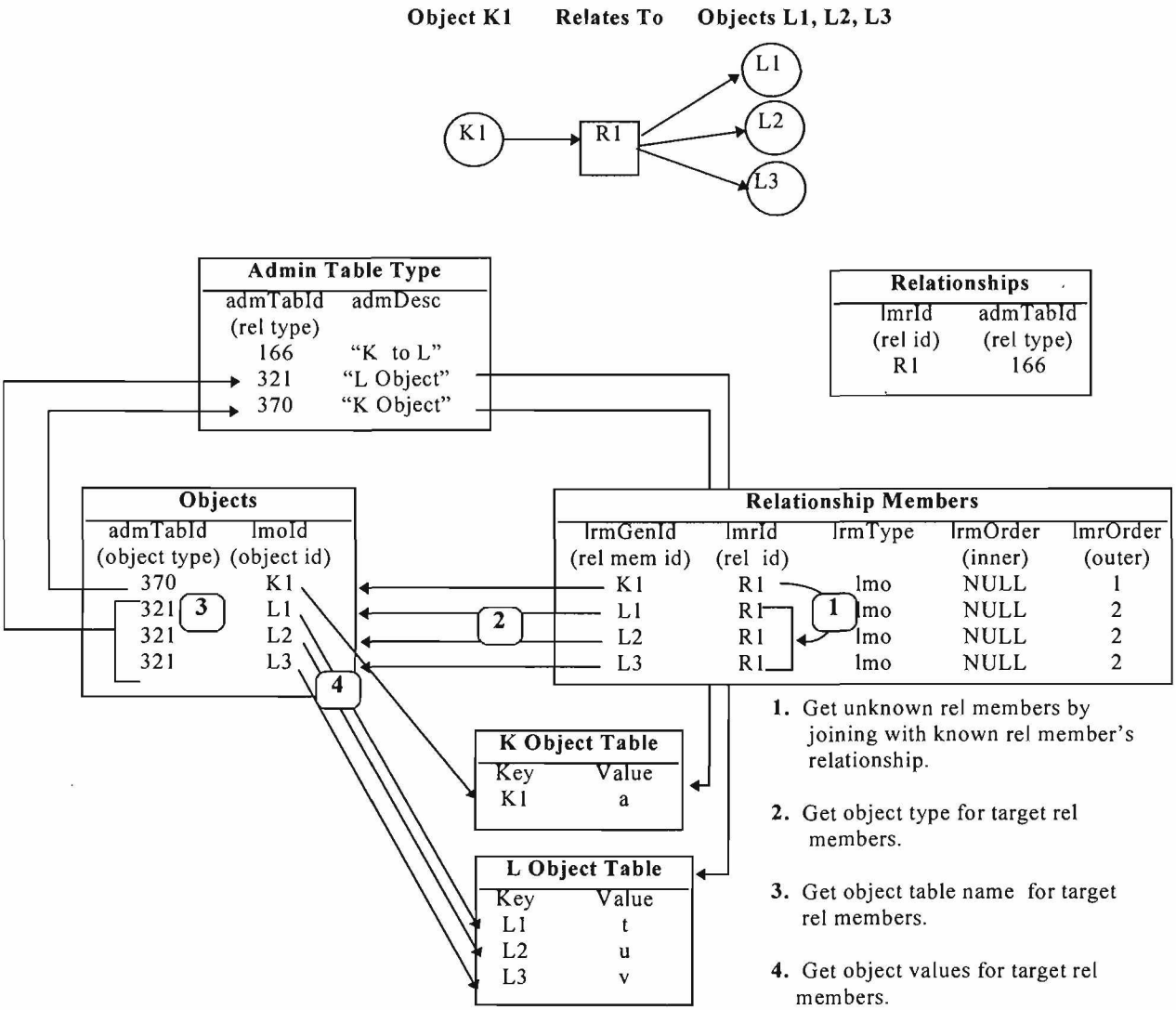
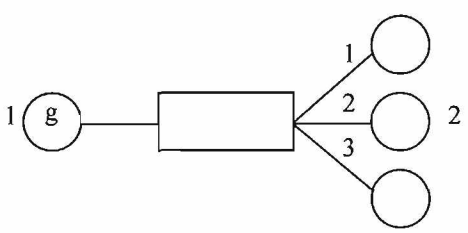


Figure 3: Relationships as implemented in Sybase.

Much of the information in a relationship is contained in the order numbers. For instance a one to many relationship is represented by a single item with major order one and many items all with order number two. Let's see how his would work in an application. The simplest case is where the object has only one relationship to traverse and wants to iterate through the many objects on the other side:

```
object g( ... );
collection & c = g.get_side( 2 );
```



1.3 Processes

Processes are the execution of a protocol. The set of processes performed in the project are organized in a "pipeline" with one process taking input from some upstream generator, computing on its inputs, generating outputs which are used as the inputs to another process. Each process is managed by a "process manager" which consists of the following sequence of steps:

1. The process manager selects an item from its input queue. (This causes that item to be unavailable to any other process manager for dequeuing.);
2. The process manager creates a process;
3. The process manager passes the inputs to the process;
4. The process begins computing;
5. The process manager monitors/waits for the termination of the process;
6. When the process terminates the process manager reads status and output information from the process;
7. Depending on the status, the following may occur:

Catastrophic failure — requeue the item with error information and a "hold" status;
Data Problem — reroute the item to some other process;
Normal — route the output to the next process.

2. Class Descriptions

Although the title of this section is "Class Descriptions" we are not discussing C++ classes, but concepts which are the focus of computing. A goal of this work is to describe a software system referring to the implementation only when necessary to ensure plausibility. Unfortunately, this is quite difficult so instead we will assume a language with roughly the features of C++ and attempt to rely on these features as little as possible. The descriptions that follow should be implementable in languages such as Objective C, C++, Common Lisp Object System, Java, and maybe [Incr TCL].

2.1 Support Types

2.1.1 Gorp

The class names described in the remainder of this section are encapsulated within a name space. This space is labeled "Gorp".

2.1.2 Access (Abstract Base Class)

An Access object contains sufficient information for an application or class to access objects of interest. In general, the access method for an object will include an Access whenever objects are accessed. Access objects can be passed from one object or process to another. Typically an Access will contain information sufficient to logon to some service. The service will then return a token used in subsequent references to the service. Examples might include: logging onto Sybase and receiving a `DBCHANNEL` pointer in return; opening a file and receiving a file descriptor; accessing a remote site through sockets and receiving a socket descriptor. There may be no return token; instead, the user name and password may be used on each access. Since each of these services requires different data an Access should be treated as an abstract base class whose derived members contain the application specific information.

The Access object also bundles the thing to be accessed (i.e., a context) with the user name and password. In the above examples we would bundle the Sybase database and dataset names, or the host and file name, or the host name and port number, respectively. Since an Access represents the right to access some service, the derived Access object itself may perform the appropriate authentication during construction or may wait until the Access is used by an application.

Interface:

```
Create( const String username, const String password )
virtual Destroy()
virtual Boolean operator=( const Access )
```

2.1.3 HandleID

An ID uniquely identifies an object. A HandleID uniquely identifies an Object, Relationship, Process, or Protocol. The HandleID is an opaque[1] type whose only operations are request , and compare for equality or order. It *should not* be assumed that this type has any particular size, is compact, can be converted to an integer, or any other implementation defined characteristic. Strictly speaking it is not possible to convert a HandleID to any other type (e.g., integer). All entities that have a HandleID live in the same namespace, i.e., given a HandleID there is one and only one entity with that ID.

HandleIDs cannot be created or deleted. HandleIDs are allocated from an infinite pool of IDs which are never reused. A HandleID can be requested or reserved from this list but is not "created", nor can HandleIDs be destroyed or deleted.

To aid applications using HandleIDs we can define an ordering which allows HandleIDs to be used as key in run-time tables. It should not be assumed that this ordering has any semantic significance other than as a convenience for lookup.

There must be some central authority which allocates HandleIDs. This might be the persistent store. Each instance of a Handle must have a unique ID (allocated by the authority) regardless of whether the instance actually persists or not.

Interface:

```
request()
Boolean operator=( const HandleID )
Boolean operator<( const HandleID )
```

2.1.4 Debug (Abstract Base Class)

Maybe this class should not be here, but I do think debugging and development support should be part of a software design, so here it is. This class is intended to provide facilities to every other class in the system. As such, this class might be the root of a class hierarchy or it might be that all classes multiply inherit from Debug. The basic service it provides is a `debug` function that prints the object and all its contents. The advantage of having this class as the single root of a hierarchy is that the function can be called easily from a debugger using a simple function wrapper.

We might also discuss here support for run-time type information (RTTI). The C++ language definition supports a particular syntax for RTTI which most compilers do not yet support. This feature and syntax can be supported through a singly rooted type hierarchy and some auxiliary functions. RTTI is so useful that I would not consider building Gorp without it. The only question is what is the best implementation. I have an implementation that is currently being used by other projects.

---

[1] An opaque type is one whose implementation is not known by the application. We are familiar with hiding implementation for classes since there is a protection boundary, but not so familiar when basic types (like enumerations) are used. It is important that a GorpID not be confused with integers.

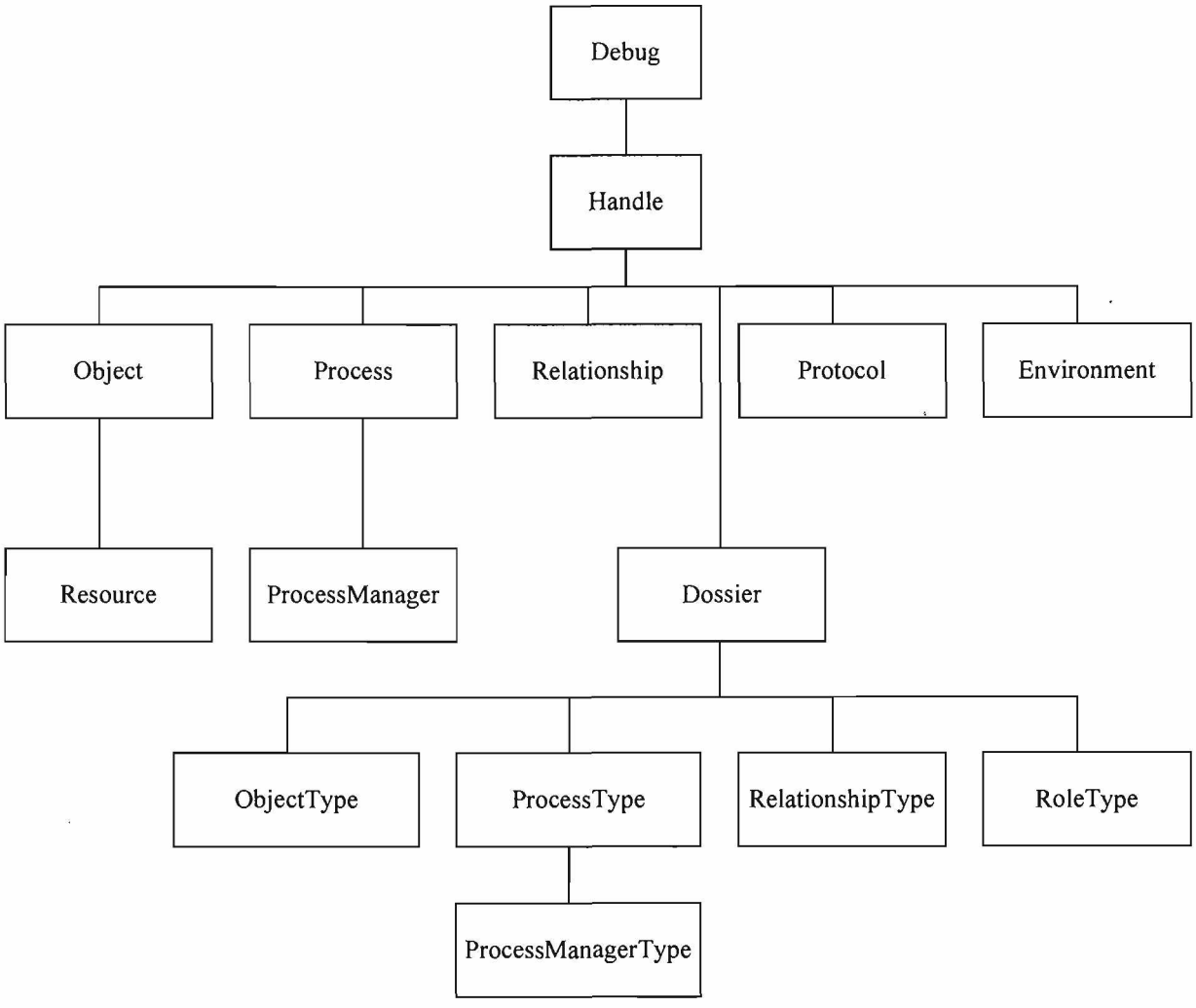


Figure 4: The Gorp Class Hierarchy.

```
virtual void debug( unsigned lvl = 0 )
```

2.2 Object Model

The object model attempts to be as faithful as possible to the Model while adding as much value due to the OOPL as possible. A basic outline of the object hierarchy is depicted in Figure 4.

2.2.1 Handle (IsA Debug)

A Handle is the basic type of all things in the Model. Handles embody state:

```
HandleID                id_
Dossier &               type_
String                  name_
String                  description_
static Access           auth_
```

That is, every Object, Relationship, Process, Protocol, and Environment in the system has an ID[2] which distinguishes this instance from every other instance. Note that although Environments do not need their own ID, because they have a one-to-one relationship with Process instances, the meta information associated which each Handle requires it. In addition, each of these can have a name and a description. These are not required, however.

The Strings name_ and description_ cannot be shared between Handle instances. The value of description_ is intended for human use only. No programmatically significant information can be stored in a description_. The name_ and description_ members of a Handle label and describe the Handle *instance* of which they are members, they do not label or describe the type of the instance. That is performed by the Dossier.

Conceptually, Handles are abstract base classes in that there will exist no Handles, but only objects derived from Handle. In reality, however, it will be very convenient to allow Handle instances to be allocated, stored, and computed upon. This allows lazy loading of object information. Therefore at run-time there will exist instances of Handle and instances of classes derived from Handle. Note that every Handle instance is incomplete in that the value associated with the HandleID is always of some derived type. This creates an opportunity for confusion when a function has a reference to a Handle, it may refer to a fully instantiated instance of the derived class or a "sliced" version of the class consisting only of a Handle. Programmers might be tempted to view the Dossier of a Handle as knowledge which provides implicit permission to cast from a Handle pointer to some derived type. As the previous example shows this assumption is unsafe. C-style downcasts and this use of Dossier are explicitly disallowed. The only legal techniques for converting from a Handle to a derived class are either through the use of a derived class constructor or through the `dynamic_cast` operator of the RTTI system. **{Issue: Is there anyway to prevent the use of this type of illegal cast?}**

What can applications do with Handle instances? An application can read the state associated with a Handle. The more operations available to applications, the more heavy weight Handles become and the more loose types become. Here are some candidate operations we might support: destroy instance, inquire derived type, copy, debug (i.e., print my contents). These are relatively simple to implement. Another operation should be "commit to persistent store". This operation would be dynamically bound to allow the derived class to control the operation.

The Handle class has a static instance of an Access which must be valid before any instance of a Handle or its derived classes can be instantiated. This is because each Handle requires type information which describes it to be loaded from the persistent store and each instance has a unique ID loaded from the persistent store. There are only two ways to provide this authorization: a "local" variable, or a "global" variable or function. The first technique requires an Access argument to every constructor derived from Handle. The second technique suggests a static data member (since an actual global variable is much worse and unnecessary). This does imply, however, that most applications use a single channel to the persistent store to perform their work.

Interface:

```
virtual void      destroy()
virtual Handle    copy()
virtual Boolean   commit()
static void       set_access( Access & a )
```

In addition to Handle, we will want collections of Handle references. These collections could be ordered, unordered, indexable, etc. To traverse these collections we should have a set of iterators which hide the details of sequential access.

---

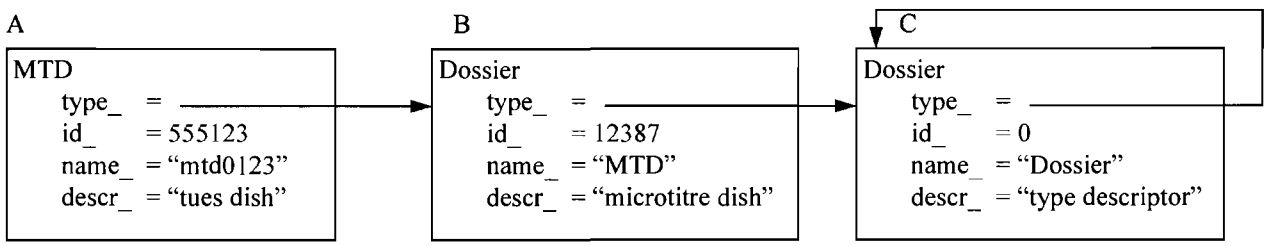
2 This is true whether the instance is persistent or not. The computational cost of this can be made quite reasonable by allocating IDs to processes in blocks of 100 or 1000. The cost of not providing all instances with unique IDs is that functions manipulating Gorps cannot depend on a "unique" identifier. That is, in a running application a single persistent object will be represented by many distinct object instances. Some of these will be "sliced" Gorps, some will be fully instantiated leaf classes. The only way a function can identify two Handle references as representing the same persistent object is by the instance ID member.

2.2.2 Dossier (IsA Handle)

Each entity that has a HandleID also has a type which is represented as an instance of Dossier. All objects of a given type share their Dossier within a single application. The Dossier for an object is an instance and has a HandleID. Thus, HandleID also serves as the type identifier for Handle classes.

An object can identify its type to an application by returning a reference to its Dossier or by returning the HandleID of its Dossier. This returned ID should identify the most-derived type for which a Dossier is available (i.e., "MTD" not "Object"). This can be implemented through run-time type information for all objects derived from Handle, for instance. There should be a one-to-many relationship between Dossiers and classes in the OOPL. Specifically, a Dossier identifies to a class with possible derived classes. Any type which has persistent meta-information should have a Dossier instance.

The type_ member of a Handle is a reference to its Dossier available at run-time. The value of the id_ member in a Dossier is the type identifier for that type (i.e., the admTabId in admTabType). The name_ in a Dossier is the name of the type, and the description_ is the description of that type. The type_ member of a Dossier references a universal Dossier. This is can be diagrammed like this:



The Dossier instance named "Dossier" is the type descriptor for all Dossiers. It is unique in that its id_ is zero, its type member points to itself and there is only one instance of this in any process. Note that the instances labeled "B" and "C" always refer to the actual type of the persistent "A", not the "sliced" type of the instance in memory at any moment. That is, even if the actual type of "A" is Handle, "B" will still have the id_ "12387".

A Dossier can have derived classes if they are necessary to describe the type in question. At the time of this writing we see a need for RelationshipType, RoleType, ObjectType, ProcessType, and ProcessManagerType classes. These five are described later in this document.

If types have unique IDs, there might be a tendency to enumerate each of the types available in the persistent store in the source code. This would lead to the problem that adding a new type would entail modifying source code with the resultant recompile. This would defeat one of the goals of the design. Instead, it is reasonable to define a type without enumerating its value. For convenience, those types which are referenced often in applications should be declared as symbolic constants in the source code. These symbolic constants should be centrally located.

If inserting symbolic constants for relationships is objectionable a compromise would provide a special member function accepts a String and performs a run-time query against the persistent store to determine the corresponding Dossier.[3] This would also require some type of exception handling if the string did not correspond to an actual Dossier in the store.

---

[3] The distinction between an enumeration, a cpp define, and a string is misleading. The actual issue is where are symbolic names and type IDs located and when are they bound? There are three possibilities: (1) in C++ header files; (2) in C++ source files; (3) in the persistent store. For the first two choices either enumerations or cpp defines can be used, but for choice (2) we then have the problem of ensuring that multiple symbolic definitions of the Dossier use the same symbol. That is, copying the definition of a symbol into source files is simply poor programming practice. This leaves choice (1) which is distasteful to some because it solidifies the fluid nature of relationships in the persistent store. To address this the third choice seems a reasonable compromise since we can associate symbolic names with Dossiers and perform the mapping of those names using data in the persistent store. Since the store does not support a "symbol" type we are forced to use array of char for this purpose.

The intent of the Gorp design is that all operations on the persistent store be carried out through these classes. This includes persistent store maintenance and administration. For the moment, however, the document describes only the application interface with the understanding that this interface is incomplete.

2.2.3 Relationship (IsA Handle)

A Relationship is a collection of Roles. This collection is unordered (ordered collections are disallowed, see Role for an explanation). A Relationship can contain zero Roles, but only Relationships with one or more Roles can persist. Roles are identified by a Dossier containing an optional associated string name. Within a single Relationship the string names of Roles should be unique.

A Relationship has a Dossier whose ID corresponds to a `reltype` in **Figure 3** and represents a named relationship in the Model. The Relationship contains references to all Roles participating in the relationship. The basic Relationship corresponds to a "thin relationship" in the Model, and derived classes correspond to "thick relationships". A query on the Relationship class can return an iterator over Role or, for convenience, can return an element of a Role. Some possibilities for the query interface are:

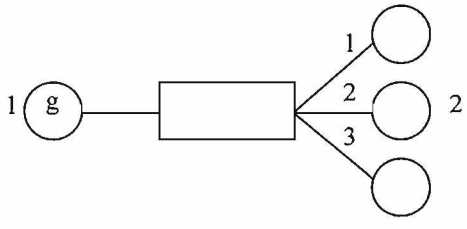
```
Iterator &  get_role( Dossier id )  // Return this role.
Iterator &  get_role( String name ) // Get this named role.
Handle &    get_item( HandleID id ) // Return an item by ID.
Handle &    get_item( String name ) // Return an item by name.
```

Relationship instances can have names and descriptions. It does not appear that these members will be used often. The Dossier associated with a Relationship also has a name and description. The name and description embedded in the Dossier of a Relationship instance should be used to access Relationships since this is the application-visible name of the relationship.

The current concept of major (or outer) order number is replaced by the Role and its Dossier instance for the many-to-one relationship in the previous section, we might use the following code:

MTD m = ...;

Iterator & iter = m.get_role(2);



This is a poor interface because inserting raw numbers is error prone and difficult to maintain. Instead we can use the name in the Role Dossier which is persistent and can be used instead of order numbers in applications. The string name associated with a Role should relate to the *meaning* of the role rather than the types involved. The above example would then look like:[4]

```
MTD         m = ...;
Iterator &  iter = m.get_role( "wells" );
```

Following a "pointer" chain might look like:

```
Cosmid & c = dynamic_cast<Cosmid &>( iter.first().get_item( "cos142" ) )
```

Note that the dynamic cast is necessary since get_item returns a generic Handle reference which cannot be treated as a derived type. If these named relationships are "permanent" in that they will likely not be removed in the future, a more convenient interface can be implemented with wrapper functions:

[4] Note that all the above examples are simplified in that a Object participates in many relationships and that before accessing the `get` functions we must first find the proper Relationship with a `get_relationship()` call.

```
Well & Gorp::MTD::get_first_well()
{
    return dynamic_cast< Well & >( rel_.get_role( "wells" ).first() );
}
```

Convenience functions such as this would be defined in a derived class of Relationship. This removes some flexibility from the design, but the convenience to applications would be of immense value.

Relationships can participate in relationships. This can be accommodated by adding a set of Relationships as a data member to Relationship. This is necessary because the relationships it participates in are distinct from those relationships that the instance describes. The interface to this relationship is identical to that used by Object:

```
Iterator &          get_rel_by_type( HandleID type_id )
Iterator &          get_rel_by_type( String type_name )
Iterator &          get_rel_by_type( Dossier & type )
Relationship &      get_rel_by_inst( HandleID instance_id )
Relationship &      get_rel_by_inst( String instance_name )
```

State:

```
RoleBag[5] *        roles_
RelationshipBag *   rel_
```

Interface:

```
create( HandleID );
Iterator &          get_role( Dossier t )
Iterator &          get_role( String name )
Handle &            get_item( HandleID id )
Handle &            get_item( String name )
Iterator &          get_rel_by_type( HandleID type_id )
Iterator &          get_rel_by_type( String type_name )
Iterator &          get_rel_by_type( Dossier & type )
Relationship &      get_rel_by_inst( HandleID instance_id )
Relationship &      get_rel_by_inst( String instance_name )
```

2.2.4 RelationshipType (IsA Dossier)

A RelationshipType is a type descriptor for Relationship. There is only one additional member provided by RelationshipType:

```
RoleTypeBag *       roles_
```

That is, a RelationshipType defines the type identifier (i.e., HandleID) for a class of relationships, their relationship name and description, and the set of Role types defined on a relationship. This information is available to applications which have either an instance of the Relationship or a handle on the RelationshipType. Since this information is persistent it is reasonable to allow applications to inquire about the characteristics of a relationship even if no instances are handy. These RoleType instances are also referenced by the type_ data member of the Roles with that type.

The treatment of name_ in RelationshipType is interesting. This value does not hold the class name of the Relationship instance, rather it contains the logical name of the relationship (e.g., "CosmidXWell" versus "Relationship"). The broad class of an instance (i.e., object, relationship, process) can be determined by using run-time type information on the instance's dossier. For example:

```
void foo( Handle & h )
```

[5] We have introduced a collection class called OrderedBag which is an ordered collection allowing duplicates. These bags manage their own memory and are implemented with templates.

```
{
        if ( dynamic_cast< ObjectType & >( h.type() ) )
              // It is an object.
        else if ( dynamic_cast< RelationshipType & >( h.type() ) )
        {
              // It is a relationship.
              // Now determine what type of relationship.
              StringType r = h.type().name();
        }
        else if ( dynamic_cast< ProcessManagerType & >( h.type() ) )
              // It is a process manager.
        else if ( dynamic_cast< ProcessType & >( h.type() ) )
              // It is a process.
        else
              // Something else.
}
```

Of course, this idiom can packaged more neatly.

2.2.5 Role (IsA Handle)

A Role is a collection of Handle references. This class represents one "side" of a Relationship. The collection can be represented as unordered or as a vector with a integer subscript. It can contain zero references, but only Roles with one or more references can persist. The members of a Role must be of a single type (i.e., the Dossiers of all members must have the same ID). There is no support for naming the members of a Role.[6]

```
HandleBag *        members_     // Either an Object or Relationship
```

The ordering of the members in a Role indicates that the i^{th} member of the role has some computable relationship with the $i^{th}+1$ member. This means that more members can be added to the role and that their relative order is guaranteed to remain unchanged although their index may change. In other words, the index of a member in a role should never be used to ascribe special meaning to the element.

> In summary, roles in a relationship can be named, but not ordered, and items in a role can be ordered but not named. This provides all the necessary structuring facilities within a clear and simple framework.

All of the data members have public read-only accessors. Since there is no member of Relationship which returns a Role, these accessors are effectively private to Relationship. That is, this class is an artifact of implementation and is not visible to an application.

2.2.6 RoleType (IsA Dossier)

A RoleType is a type descriptor for Role. This class has these members:

```
Boolean      vector_of_
Dossier      member_type_
```

which indicate that the Role is to be treated as a vector or as an unordered set[7]. The members of a Role must all be of the same type. That type is recorded persistently in the member_type_ variable. The name_ member of a RoleType (in Handle) describes the logical grouping, which is typically not the type of the things grouped, rather than the class name of the instance (typically "Role").

---

6 The purpose of ordering collections whose members are of different types is to implement positional parameters analogous to arguments to shell scripts. Modern programming languages avoid naming positional parameters by numbers and allow symbols instead. Using symbols simplifies the model of relationships and improves clarity.

7 This information can be stored persistently in a RoleType table. Since RoleType IDs are unique an intersection table can relate reltype to RoleType ID.

2.2.7 Object (IsA Handle)

An Object represents physical things. For instance, anything that can be barcoded is an Object. Objects have an unordered set of Relationships. These relationships can change during the lifetime of a single Object instance and reflect the persistent set of relationships in which the object participates. Relationships can be referenced by any of their characteristics:

```
Iterator &          get_rel_by_type( HandleID type_id )
Iterator &          get_rel_by_type( String type_name )
Iterator &          get_rel_by_type( Dossier & type )
Relationship &      get_rel_by_inst( HandleID instance_id )
Relationship &      get_rel_by_inst( String instance_name )
```

Most of the issues regarding accessing dynamic Relationships applies to accessing the relationships of an Object and the same solutions apply as well. To identify a particular relationship within an object we use the following technique:

```
Object       g = ...;
Iterator &   gr = g.get_rel_by_type( "CosMtdxCosWell" );
Object &     wr = gr.get_member( well_id );
```

2.2.8 ObjectType (IsA Dossier)

An ObjectType is a descriptor for Object. Its data member is a set of the RelationshipTypes the Object participates in:

```
RelationshipTypeBag * relationships_;
```

2.2.9 Process (IsA Handle)

The Process class represents the execution of a protocol. The PM manages collections of Processes through the ProcessSet class. Applications derive from Process to provide an interface to their inputs, outputs, and PM. The state in a Process is:

```
ProcessStatus           status_
Priority                priority_
Access &                access_
IOItem &                io_
OrderedSet< time >      time_
Environment             env_
```

The Access is used by the Process to manipulate external resources, including acquiring its inputs and routing its outputs. The IOItem is actually a reference to an derived class: QueueItems are referenced by Processes in the PM, AppIOItems are referenced by application derived class Processes.

The time member is used by the ProcessSet class to allow scheduling of processes for future execution, recording the starting time of running processes, and recording the termination time of finished processes. It is a set so that a completed process can include both its starting and ending times.

Interface:

```
virtual ProcessManager  get_manager()
Boolean                 route( Set< Set< Handle & > > outputs )
```

A Process object has two incarnations: a generic one which lives in the PM and is used to manage processes, and a derived class version which lives in the application and embodies the data and behavior specific to that application. Typically, each application will derive a new class from Process which contains all the data specific to the application. Behaviors for that application (e.g., what it does, its api) will be declared as member functions of the

class. Applications retrieve their inputs from the persistent store by first requesting the IOItems from the Process object, then instantiating complete input objects using a derived class of IOItem (see 2.2.15.3).

2.2.10 ProcessType (IsA Dossier)

A ProcessType (PT) is a type descriptor for Process. In addition to the Handle members PTs have:

```
DossierBag              inputs_
DossierBag              outputs_
Protocol                protocol_
ResourceBag             resources_
EnvironmentTypeBag      environments_
```

The input to a Process is an ordered set (outer) of ordered sets (inner). Each outer set contains a specific type, but any number of members. For example, a Process can be defined to accept a set of images and a set of microtitre dishes. Each set of inputs will then consist of one or more images and one or more microtitre dishes. Likewise each execution of the process produces a set of sets as output with the same constraints. Only the basic types of these input sets are actually recorded in the class.

The resource list is used to determine if a Process can be executed once its inputs are available. To accomplish this we must know what resources are required. In a more elaborate scheduling system we would also like to know for what duration each resource is required. Ultimately the resource list should contain a set of ResourceQuantitys.

The last element of a ProcessType is the environmental and instance values which should be recorded for each execution of the Process. This requires some kind of identifier for each item. An example of these might be:

Person	Date	Time
Resources used	Protocol	Changes to Protocol

The precise value of this attribute needs more work.

2.2.11 Environment (IsA Handle)

This class holds the records of process execution. It is intended to contain any information which varies from execution to execution and which is not contained in the Process records (such as inputs, outputs, protocols, etc.). The data recorded can have structure and can be recorded hierarchically:

```
UnorderedBag< pair< String, String > >    values_
OrderedSet< Environment * >               env_
```

That is, an environment record is an ordered collection of strings which denote attribute value pairs. Environments are hierarchical in that an environment can consist of a sequence of environmental readings. For instance, if an environment consists of a set a base values along with a time-varying sequence of values the base values would be stored in the top-level Environment with the time-varying readings stored in sub-Environments, one per time sample.

2.2.12 Protocol (IsA Handle)

A Protocol is a hierarchical representation of a set of steps. Each step in a protocol may be a reference to a resource or other Protocol. Protocols also include:

```
StepBag     steps_
String      summary_
String      references_
```

{Issue: how do we do versions?}

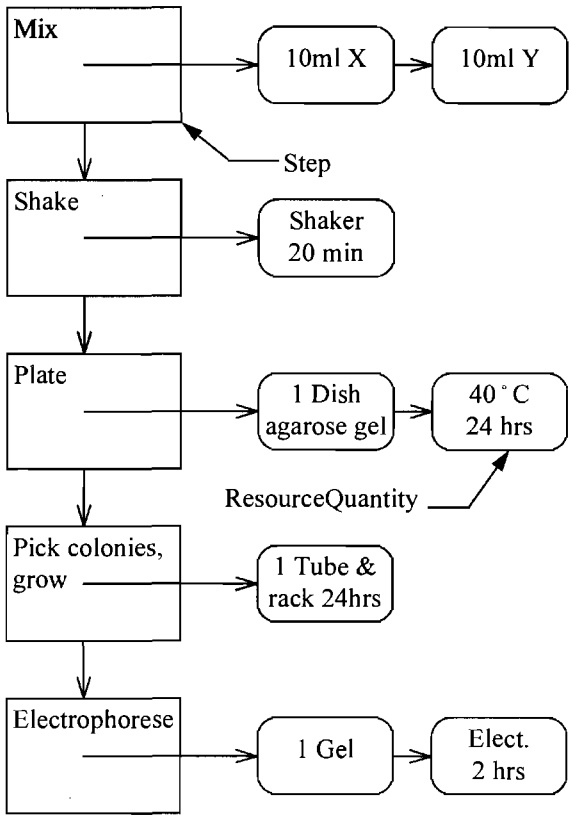


Figure 5: Example use of Steps.

2.2.12.1 Step (IsA Debug)

Protocols consist of ordered sequences of Steps. The state in a Step is:

```
String            instruction_
GorpTime          duration_
Protocol          protocol_
ResourceQtyBag    resource_
```

The protocol_ and resource_ data members are mutually exclusive in that one or the other may be a valid reference, but not both. A Step which is a reference to another Protocol cannot have a resource list of its own. Rather, the target Protocol has a resource list.

Steps are sequential and each step can have a duration. The duration recorded in a step has no direct relationship to the amount of time consumed by various referenced ResourceQuantitys. That is, ResourceQuantitys in the resource list may indicate the resource is in use for a certain duration. The sum of the durations in the resource list may be larger or smaller than the duration recorded in the Step.

Protocols are inherently hierarchical. Any stage of a protocol may consist of another Protocol. Each step of a protocol is completely described (to humans) by the comment field of a Step. If a step consists of other steps they are referenced by the elements_ member of the Step. A step of a protocol may rely on or consume resources. These resources are recorded in the resource_ member. For example, assume the fictional protocol below:

1. mix 10ml X with 23ml Y
2. Shake for 20 minutes
3. Plate out on agarose gel and incubate for 10 hours at 40 ° C
4. Pick colonies
5. Grow colony in medium for 24 hours
6. Electrophorese 2 hours

This protocol might be represented by Steps as shown in Figure 5.

2.2.12.2 ResourceQuantity (IsA Debug)

ResourceQuantitys are used by Steps to enumerate the type and quantity of resource consumed. They contain a reference to a resource, and an indication of time, weight, or volume required by the protocol step. ResourceQuantitys are linked together in an ordered bag.

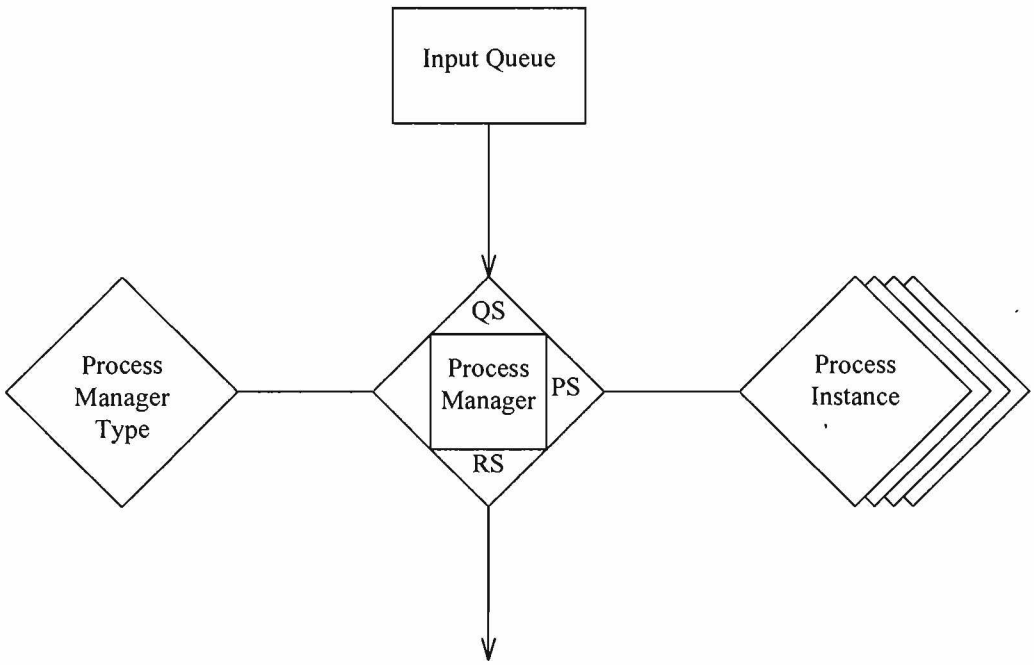


Figure 6: The ProcessManager Diagram.

2.2.13 ProcessManager (IsA Process)

A ProcessManager (PM) is a class for creating, destroying, controlling, and querying Processes. An instance of a PM controls a single type of process (i.e., Dossier). The fundamental behavior of a PM is defined by ProcessEvent (described below). PMs have the following attributes:

```
IOQueue          queue_
ProcessSet       processes_
Router           route_
```

Each of these classes is described below. Briefly, a PM controls an input queue of items to be processed, a set of Processes some of which have already run, some are running, and some are scheduled to be run. The output of Processes is a status and set of Handles which are routed to the input queue of other PMs. ProcessTypes are used as templates when instantiating Processes.

Although a PM controls a single type of Process, it is possible (and reasonable) for a single Process type to have more than one PM instance, each with its own input queue and router "located" in different places in the pipeline. This allows PM to be treated as modules to be replicated as needed. The PMs are distinguished by their HandleID.

A particular PM can be instantiated many times in various processes and all instantiations with the same HandleID represent the same PM. The actual state of the PM is stored persistently and each instance interacts with the persistent state through a controlled interface. This transforms the problem of requiring a single PM to manage a shared queue in a distributed, unreliable computing environment into the simpler problem of managing a shared memory object. Another consequence is that, for small amounts of communication, a PM and a Process can communicate without Unix-style Interprocess Communication (IPC). Rather, the target Process can instantiate its PM and request the input values through direct member function call. If an application has a fully functional PM in its address space, then the transmitting of a return status and HandleID list to the PM for recording and enqueuing can be performed the same way, through the persistent object itself.

Communication between a PM and its Process can be performed in the following ways:

1. Command line arguments;
2. Environment variables;
3. Pipes such as stdin and stdout;
4. Sockets;
5. Persistent store.

Any combination of these may be employed by a third party application, but all applications written locally should use the last technique. Arguments to an application can be bundled as a Handle (or derived class) and stored persistently. The arguments are then placed in a PM's input queue and the application can then retrieve them through direct calls on a locally allocated PM. Third party applications cannot use this technique without a locally written wrapper. Such a wrapper might be made generic enough to be used by all third party applications, but adding this support directly to the PM is a reasonable alternative. **{Issue: How much third party app support should be in the PM?}**

Applications must interact with the PM to acquire their inputs. Here is a sample of that interaction:

```
LaneFinderProcess &     lf;
Image *                 ip = lf.get_image();
```

An application first acquires a handle on its Process object. The Process object is then used to access the application's input data. Here are some details. When a PM dequeues a QueueItem and creates an instance of the Process, this is recorded persistently:

Process HandleID	Dossier	Process PID	Host	PM HandleID

In the application the LaneFinderProcess constructor uses its PID and host name to find the PM HandleID, instantiate the ProcessManager and acquire the application's inputs. The LaneFinderProcess class has a LaneFinderIOItem member capable of instantiating the proper input types from the Handles returned by the ProcessManager.

During the execution of an application new objects may be created and committed to the persistent store. This commit can be performed by the application objects themselves rather than by a PM, Process, or IOItem object. The IDs of the committed objects must be transmitted to a PM for routing to the next PM in the pipeline. This is done by invoking the `route` operation on the LaneFinderProcess object. This member function accesses the PM and uses the `enqueue` operation of the class. Given the current design of the PM *it does not matter whether this enqueue operation is performed by the PM in the application or some other PM.* By performing the operation in the application we can avoid the issue of IPC as we did with application inputs.

ProcessManager instances have a member function to allow enqueuing items. This function is used by the Router to move objects through the pipeline. The enqueue operation is type-safe in that only certain types are legal for enqueuing. This type checking is performed at run-time by accessing the list of legal input types from the ProcessType class (which may acquire this list by checking the administration tables in the persistent store).

A PM must be prepared to capture signals generated by Processes it controls. In particular, if a Process dies the PM must catch the signal, diagnose the problem, and respond appropriately. One appropriate response is to find the input QueueItem, attach an error report to it (see QueueItem) and requeue the item. If the item fails repeatedly, the PM might change the status of a QueueItem to "hold" before requeuing.

There is some concern that whenever an application instantiates a PM it has full control over the state of the PM. This can be addressed by using the PM as a base class all of whose services are protected. Two derived classes are

declared: SProcessManager and CProcessManager (server and client, respectively). These classes make a subset of the protected functionality public. Applications allocate a CProcessManager while demons and controllers allocate an SProcessManager.

{Issue: Can we work some dataflow concepts into queues and routing?} This is a serious issue. Given that a process may generate an object which should be enqueued in an arbitrary PM downstream and that the input for that PM may require objects which have not yet been created, how is the rendezvous between these two objects accomplished?

The PM class can be made entirely generic by providing a process create function table whose key is the ProcessType handle id. Other process specific behavior, such as routing, acquiring process instances, type checking queue items, can all be data driven from meta data in the persistent store.

2.2.14 ProcessManagerType (IsA ProcessType)

A ProcessManagerType (PMT) is a type descriptor for ProcessManager. It contains a single additional data member:

```
ProcessType        managed_process_type_
```

This is a reference to the Process dossier to allow the manager to check input types to the queue and identify resources required for scheduling.

2.2.15 IOQueue (IsA Handle)

A PM manages a queue of input items defined by the IOQueue class. An IOQueue is a priority queue where elements are added to the tail and removed from the head. The goal is to allow the PM to select the next QueueItem to process under all circumstances. That is, the user is viewed as a cooperative member of the workflow, not as "master" of control. This also allows (human) managers to set the order in which work is performed at a reasonable level of detail. Elements in the queue are ordered by a process specific algorithm. A default ordering algorithm orders elements by priority then time of entry. IOQueues consist of QueueItems each of which represents a unit of work waiting to be performed. Each IOQueue is owned by one PM which has sole control over the IOQueue. IOQueues contain QueueItems which collect all inputs to a single execution of the Process.

Under special circumstances it is desirable to allow users to request an "out of order" item to be dequeued next, to a certain extent "ignoring" the order of the queue. This can be addressed by allowing QueueItems to be reprioritized and resorted. If queues are handled as a shared resource this would require the following steps: (i) lock the queue; (ii) reset QueueItem priority; (iii) sort the queue; (iv) remove the head of the queue; and (v) unlock. This handles the race condition of a user reprioritizing the queue and having some other process remove the item before the original user can get to it. **{Issue: Synchronization needs to be discussed more thoroughly.}** Users can only remove the head of the queue, period. A user may be allowed to reprioritize elements in the queue, but that does not guarantee the reprioritized element will be at the head.

What operations can be performed on a queue? Certainly the following:

- Enqueue an item,
- Dequeue an item,
- View the queue,
- Sort the queue,
- Lock and unlock the queue,

Viewing the queue can be made to encompass all types of queries, however, common operations such as determining queue length, highest priority item, longest wait time, etc. might be implemented as explicit operations.

Queues cannot be halted, this effect is achieved through the PM.

It is not possible to change the prioritizing algorithm of an IOQueue after construction, mainly because there is no need for this feature at this time.

Since queues control the work flow there is a need for restricting access to them. We propose that the PM be the focus for authorization and authentication. If a process can acquire a valid PM handle, then all operations are then available.

An IOQueue can be made responsible for actually performing what ever persistent manipulations are necessary to manage the queue. This would require providing an IOQueue with an Access to access the persistent store and an identifier for the Process which owns it.

2.2.15.1 IOItem (IsA Debug)

An IOItem is a base class **{Issue: Abstract base class?}** which contains:

```
HandleBag   inputs_
HandleBag   outputs_
```

The class is used by the IOQueue to create a queue of input items and by an application specific IOItem to access the input items and store the output items. The goal is to allow a PM to dequeue an item, pass it to a Process for accessing its inputs and storing its outputs, and receive a generic item as the output of the Process. See the derived classes for a more complete picture.

2.2.15.2 QueueItem (IsA IOItem)

The IOQueue managed by a PM contains a set of QueueItem instances. Each QueueItem represents a single unit of work to be performed called a *job*. The QueueItem also has a set of control information associated with its history in this particular IOQueue. The state of a QueueItem is:

```
QueueStatus             status_
Priority                priority_
Time                    time_
unsigned                request_count_
const ProcessID         enqueuing_process_
const xxx               person_
Error                   error_log_
String                  comment_
```

A QueueItem has a status whose values are: `WAITING`, `IN_USE`, and `HOLD`. The priority is a value which determines the item's place in the queue. Priority is a scale with three "sentinel" values: `LOWEST_PRIORITY`, `NORMAL_PRIORITY`, and `HIGHEST_PRIORITY`. Items are typically queued at `NORMAL_PRIORITY`, while items that have been selected "out of order" are given `HIGHEST_PRIORITY` before resorting. The time data member records when the item was enqueued. The request count indicates how many times the item has been enqueued in this queue. The error log is a cumulative log of the errors the item has encountered while in this stage of the pipeline. Each of these values can be altered by the owner IOQueue. When the priority of a QueueItem is changed the controlling IOQueue resorts the queue.

To allow accumulating status information such as request count, error log, and comments, QueueItems are not dequeued when a Process is created to handle them. Rather, they are marked as `IN_USE`, so that if an error of some type occurs the previous values are intact without requiring special handling. QueueItems are removed from the queue, when by the Router.

2.2.15.3 <App>IOItem (IsA IOItem)

This is not a single class, but a set of classes, one for each application. The purpose of the derived, application-specific class is to provide a strongly typed interface to the actual inputs to an application. The IOItem and

QueueItem classes treat application inputs as generic Handle instances with no actual derived class data. The AppIOItem set of classes take the generic Handles in an IOItem and load the actual data associated with the object when the application requests it. Thus for an image processing application the AppIOItem class would have a member function:

```
Image * get_image()
```

which would take the HandleID corresponding to the Image Dossier, query the persistent store for the image data (or file name), load the image and build an Image object for the application.

2.2.16 Router (IsA Debug)

A class whose purpose is to enqueue a set of Handles in the proper PM input queue. The last step of a PM's control loop includes receiving an exit status and some output from a Process and routing that output. The Router class performs this function. A Router is instantiated by indicating the routing table to be used. This table is defined by the type of process being controlled. An instance of Router has a member function:

```
virtual Boolean send( OrderedBag< Handle & > ids, Status s )
```

For each of the IDs in the ID list the item is enqueued in some PM's input queue. Routing can be implemented in a data-driven fashion as a table consisting of the following:

PM ID	Process Output Status	Dossier ID	Next PM ID

This technique implies that a Process can have more than one ID as output and that these IDs can be routed to different PMs. If more elaborate routing is required the send operation can be overridden in a derived class.

A Router performs the enqueue operation on the destination PM by invoking public member functions of the destination PM. To do this the Router must have a handle on the destination PM. If this handle cannot be acquired, the routing operation fails. In addition, the Router may have a handle on the destination PM, but still fail to route the object if the type of the object and the type accepted by the destination queue differ.

Interface:

```
create( HandleID process_tag );
destroy();
virtual Boolean   send( OrderedBag< Handle > ids, Status s );
Dossier &         query( Handle id, Status s );
```

The query operation returns the Dossier of the PM to which output would be queued if send were invoked on the same values.

2.2.17 ProcessSet (IsA Debug)

ProcessSet is used by a PM to manage its set of Processes. It presents an interface to the entire set of processes ever created, running, or scheduled for creation by a single PM. These processes are accessed by requesting an iterator from the ProcessSet. The different categories of process are distinguished by their ProcessStatus.

This class is an implementation artifact and is only manipulated by the PM. The PM can add and remove processes from the set. The only time processes are deleted from the set is when their scheduled execution is canceled. Processes which have completed are not removed from the set and processes which are terminated during execution are not removed.

Interface:

```
void        add( Process & proc )
void        remove( HandleID proc_id )
```

2.2.18 ProcessStatus

ProcessStatus represents the state of processes running or terminated. The status consists of two values:

```
ProcessState    state_
ProcessValue    value_
```

The state indicates the gross state of the process, for instance, running, finished, waiting. The value provides more information, for instance, a finished process might have a status of "bad termination", or "successful termination". Suggested values for state are:

runnable	running	suspended
finished	scheduled	

Possible values for the various states requires more thought. Some suggested values are:

unknown	user_abort	proc_abort
proc_suspended	proc_term	success

2.2.19 Resource (IsA Object)

A Resource is a base class for those Objects in the Model which represent laboratory items used by Protocols. Resources can be further subdivided into consumable and non-consumable resources.

Non-consumable resources typically represent laboratory equipment. They have some capacity to perform work.

- Containers have a certain capacity in volume, weight, or discreet count;
- Machines have capacity in speed (items processed per unit time), or parallelization (items simultaneously processed);

This information can be used by other software to simulate performance or schedule work.

Consumable resources typically represent materials used by protocols. These materials are either purchased or made. They can be discreet or continuous:

- Discreet materials are always consumed in fixed units, the critical information is if a suitable unit is available for consumption;
- Continuous materials are consumed in varying amounts.

2.3 Shared Structure

This design proposes that applications explicitly request objects from the persistent store through functions. This implies that the same object may be returned as the result of two (or more) distinct object requests. Does the object reference which is returned by these two calls refer to the same memory or distinct copies? To further complicate matters, it is stated in Section 2.2.1 that a single object instance may be represented at run-time as both an instance of Handle and as an instance of one of its derived classes.

This leads to a problem for the application programmer, "How many instances of a particular object do I have in memory and how do I keep them all straight?" Note that an application may acquire multiple instances of an object without "knowing" it, because object references form a complex web. Another question the programmer might ask is, "if two collections refer to the same object, how do I delete the two collections".

2.3.1 Preventing Multiple Instances

Is it possible to prevent multiple instances of an object? Given that loading Handles as representations of objects is a design fact, no we cannot prevent multiple instances across the inheritance hierarchy. Loading Handles provides too much benefit to discard it. A compromise is to minimize the number of classes in the inheritance hierarchy we

can load. For instance, load either Handles or most-derived classes (MDCs). Although this might be accomplished, it also cripples applications. For instance, I might want to select for all Relationships which satisfy some criteria. In the presence of classes derived from Relationship, forcing the application to only retrieve MDCs would make retrieving the desired objects extremely difficult. The conclusion is that we must be able to retrieve any class in the inheritance hierarchy.

Can we prevent loading multiple instances of an object at a particular place in the inheritance hierarchy? That is, given the hierarchy in Figure 7 (where each D_i is a class and D_2 inherits from D_1), can we prevent multiple instances of D_1 from existing in memory at one time? The obvious technique is to construct a table of (id, type, address) which allows the object retrieval function to determine if a particular instance which is about to be instantiated from the persistent store already exists in memory. Such a function would use an algorithm something like this:

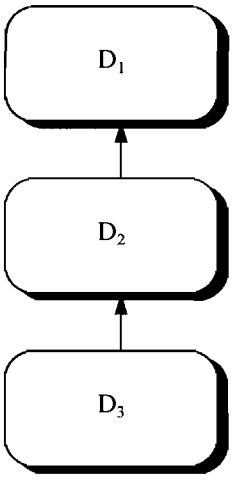


Figure 7: Example inheritance hierarchy.

1. Perform the select against the database.
2. For each object we are about to instantiate, using the HandleID check the table to determine if the object has been loaded.
3. If not, load the requested object (this might load a D_1, D_2, or D_3).
4. If so, determine if the particular *type* requested by the user has been loaded.
5. If not, we might (a) return a more derived class instance, or (b) load the actual type requested, depending on our policy.

This technique allows the persistent store to avoid returning unnecessary duplicates. The choice of what to do in step 5 is difficult. A naive policy such as "always return the MDC instance available" leads to immediate problems. For example, suppose we first retrieve a D_1, then a D_3, and finally we retrieve a D_1 again. The final retrieval would actually return a D_3 since our policy would always return an MDC where possible. Two calls on the same retrieve function would return two distinct objects of different classes. A more sensible policy is to return the requested class if it is loaded or a MDC if not. This would achieve a consistent effect in our previous example (returning a D_1, D_3, then D_1 again).

This accounts for duplicates created by the persistent store subsystem, but what about duplicates created by applications? In particular, should a copy constructor be defined for objects which exist in the persistent store? See the next section for a discussion.

2.3.2 Managing Multiple Instances

There should be a simple technique for "casting" a base class instance into a derived class instance. Since the persistent store subsystem will have a table of all loaded instances, casting from a base class instance to a derived class instance would perform a lookup and return the derived class instance if it has been loaded. If it is not loaded, the "cast" function could perform the load on-the-fly. If the base class instance has been modified, should the "cast" function propagate those modifications to the returned derived instance? If the derived instance is already known (i.e., does not need to be loaded), then it has been returned to some other caller. In this case, automatically applying the changes from the base class would cause the previous caller (who already has a reference to the derived class) to see a "spontaneous" change in the object's values!

When multiple instances of an object are resident simultaneously, there can be no reasonable way to maintain synchronization between copies. Obviously, keeping a list of all known references requires extreme effort. This means the application must manage this issue. The simplest technique being to avoid generating or using multiple instances. If an object is retrieved by the application, then mutated, subsequent retrievals will return the mutated object (unless it has been copied). This is consistent with the view that OODBs like ObjectStore use. If an

application wants the original value back again it must make a copy before the mutation. This suggests that copies should be allowed. Should the copies have the same HandleID? If we have a copy constructor which alters the HandleID, then a simple typo such as omitting an ampersand in a function call could be very hard to track down. A "copy with new id" member function would be simple to provide, however. Should same id copies be allowed?

Is a "cast" operation a constructor? That is, given an object I need to invoke some function and indicate the type which I desire. Indicating type can be done by several methods: use the type_info structure, use the type name as a quoted string, use the Dossier-derived class instance, use the HandleID of the type, or use the type name as an identifier. All but the last can be implemented as an argument to a normal C++ function or member function. They have the advantage that they can return a status. Using a constructor, on the other hand, allows applications to use the built-in cast notation, but requires exceptions to signal errors. If the cast member function is a static member in the target (or derived) class, then we can avoid explicitly naming the result type as a argument, but each class implementer must write their own cast function. A generic cast operation could be written which would require a type parameter.

2.3.3 How Do Applications Delete Instances

Who is allowed to delete an object loaded from the store and how is that accomplished? Since the retrieval function will have a table of known instances, the destruction of an instance (by some external agent) would lead to dangling pointers. The only way to avoid dangling pointers is to use some type of complete garbage collection. This might be done through a rigorous use of pointer classes or with a real non-moving garbage collector such as Hans Bohem's.

Complete and correct reference counting pointer classes are notoriously hard to build. The first step of a pointer to a single class is easy. The next step of requiring that pointers to derived classes decay into pointers to their base classes requires a complete pointer hierarchy. The final requirement that pointers to constant object be allowed mandates an entirely separate (and "duplicate") hierarchy mirroring the first hierarchy, but with constant pointer values. This quickly becomes unwieldy. Nevertheless, it has been successfully used before.

A compromise which provides an incomplete solution is the ability to delete an object once (and only once), but does not prevent dangling pointers. This is the concept of "one owner, many sharers" and can be accomplished in several ways: (a) associate ownership with the owner type; (b) associate ownership with the owned type; (c) make ownership orthogonal to type.

Associating ownership with the owner type is easy and is described in every C++ book. If an object has a reference to another object, the owner object copies or deletes the owned object in its copy constructor or destructor, respectively. The problem with this technique is that it is quite restrictive. For instance, we must have bags which own their contents and those which do not. Since they are distinct types a bag of owned things cannot be treated as a bag of shared things (or vice versa, i.e., you can't do both). Notice that there is nothing which prevents an object from being "owned" twice.

Associating ownership with the owned object would require a back pointer from the owned object to the owner. The advantage of this is that an object can only be owned by one other object as defined by the value of the back pointer. It would also be possible to follow the chain of back pointers to the root of the current structure. What is the type of this back pointer? For our purposes, it cannot be Handle since bags are not handles. If we wish to follow the chain of pointers to the root, it cannot be "void" (since we could not subsequently indirect through it). If we encapsulated the pointer and its operations in a "mixin"class we could use multiple inheritance to bundle it into any class we desire. (There is the possibility of multiple inheritance artifacts, see the next section for a discussion.) What are the operations on this mixin class?

```
bool  owned_by( Ownership * o );          // Test for ownership.
void  set_owner( Ownership * o );         // Set my owner.
void  maybe_set_owner( Ownership * o );   // Set me if I'm null.
void  get_owner();                        // Return my owner.
```

Making ownership orthogonal to type would result in some kind of global ownership table. This table could contain the address of an object, the address of its owner, and the object's type. Notice that we already have a table which maps HandleID to address and type from the previous section. By adding a single field, address of the owner, we gain the memory management features we seek. This table would be accessed through a set of functions identical to the operations required of the mixin class. In addition, this table offers the possibility of asking an object all the other objects it owns. The maintenance of this table would be managed by the objects themselves. When an instance is created its address and owner must be stored in the table. When an object is deleted, its entry must be removed from the table.

Notice also that associating ownership with the owned object alters the meaning of assignment. When an assignment occurs we have the additional decision to transfer ownership. In some sense, we have a double linked list (owner to owned) for which we may need to set both pointers.

2.3.4 Gorp and Multiple Inheritance

Gorp assumes that every instance has one and only one unique id and that this id is acquired by deriving from the Handle class. In the presence of multiple inheritance it is possible to create a class which contains more than one instance of Handle. This violates so basic assumptions and must be avoided. Notice that multiple inheritance, per se, is not at fault, it is the associating of more than one HandleID with an instance which is the problem.

3. Current Relationships

3.1 Database Inconsistencies

Instances of a given relationship type do not follow a constistent pattern either due to a variation in membership types or use of major and minor orders.

Types of relationship members cannot always be determined because of missing entries in the objects table and history table.

Inserts of lmo relationships are not being recorded in history table (the trigger has been removed for performance reasons).

Some recent instances of relationships have a "0" major order where a non-zero major order was expected.

3.2 Issues

How do we indicate an unordered set?

Are there any cases where major order on the Relationship table is still relied upon and/or used in a different way than the major order on Rel Members?

Are there instances with incomplete membership due to an incomplete transaction?

3.3 Definition of Current Relationships

3.3.1 CloneXVector

Number of instances:	13
Status:	Rel member type cannot be determined; missing from objects table.
Purpose:	This is obsolete. Replaced by process "Mate", which associates a gvo Cosmid to an lmo Vector.
Relationship:	Relates gvo Clones to lmo Vectors in lmo. This appears to be a one-to-one relationship.
Major Order:	Indicates "side". Side 1 = Cosmid (clone), Side 2 = Vector.

Minor Order: Not used.
Action: Delete.

3.3.2 CosFrgXImgBand

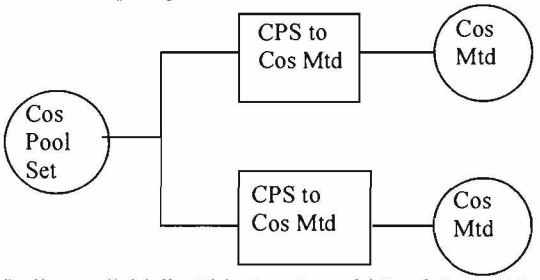
Number of instances: 5,088
Status: Consistent definition.
Purpose: ?
Relationship: Relates Cosmid Fragments to Image Bands. This appears to be a one-to-many relationship.
Major Order: Indicates "side". Side 1 = Cosmid Fragment, Side 2 = Image Bands.
Minor Order: Not used? Are Image bands ordered or unordered?
Action: Determine purpose. Determine use of minor order.

3.3.3 CosMtdXCosWell

Number of instances: 14,199
Status: Consistent definition.
Purpose: Given a Cosmid MTD Well, this relationship allows you to identify the Cosmid MTD on which this well is located. For example, this relationship is used be the Membrane Layout Screen to tie a Cosmid Well is part if a Sequence Set to a Cosmid Microtitre dish.
Relationship: Relates Cosmid Microtitre Dishes to Cosmid Wells.
Major Order: Indicates "side". Side 1 = Cosmid MTD, Side 2 = Cosmid Well.
Minor Order: Not used.
Action: Determine how this relationship is create.

3.3.4 CosPoolSetXCosMtd

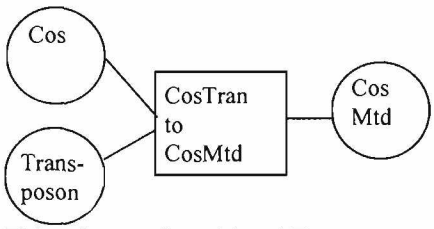
Number of instances: 1,294
Status: Consistent definition.
Purpose: Created by process "Cube Multiplex", from Mux screen. Represents a group of microtitre dishes that were muliplexed (grouped together).
Relationship: Relates Cosmid Pool Sets to Comid Microtitre Dishes. This is a one-to-many relationship implemented as a set of one-to-one relationships.



Major Order: Indicates "side". Side 1 = Cosmid Pool Set, Side 2 = Cosmid Mtd.
Minor Order: Not used.
Action: Determine if this relationship can be implemented as a one-to-many relationship.

3.3.5 CosTranXCosMtd

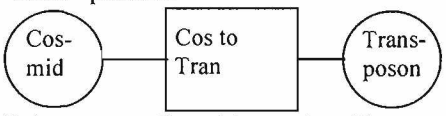
Number of instances: 1,295
Status: Consistent definition.
Purpose: Represents the pair of mated cosmid and transposon was placed on a microtitre dish by the colony picker. This relationship is created by the "Pick" process on the Mux screen.



Relationship: This relates a Cosmid and Transposon to a Cosmid Microtitre Dish. This is a two-to-one relationship.
Major Order: Indicates "side". Side 1 = Contents of dish (cosmid and transposon), Side 2 = Cosmid Mtd.
Minor Order: Side 1 may use minor order to distinguish between a cosmid and a transposon.
Action: If minor order is used to distinguish between Cosmid and Transposon, devise some other way to identify the type of object for this "side". Determine if side one should be the relationship "CosXTran".

3.3.6 CosXTran

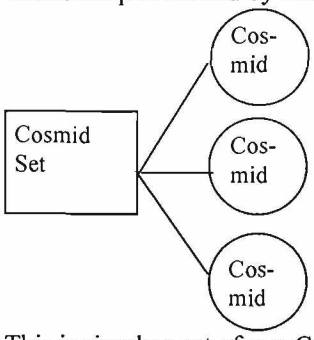
Number of instances: 342
Status: Inconsistent use of minor order.
Purpose: Indicates the cosmid and transposon that were mated. This relationship is created in the "Mate" process.



Relationship: Relates a gvo Cosmid to an lmo Transposon. This is a one-to-one relationship.
Major Order: Indicates "side". Side 1 = Cosmid, Side 2 = Transposon.
Minor Order: Not used? (First instance has minor order of NULL, last instance has minor order of 1).
Action: Check remaining entries for correctness. Replace NULL minor order with 1 where necessary.

3.3.7 Cosmid Set

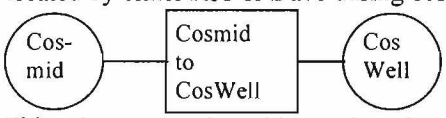
Number of instances: 53
Status: Inconsistent use of minor order.
Purpose: Groups cosmids together to be presented together from certain application screens? This relationship is created by the driver screen.



Relationship: This is simply a set of gvo Cosmids. This is a one-sided relationship.
Major Order: Currently indicates "side", but this is not necessary since this is a 1 sided relationship.
Minor Order: The last instance of this relationship appears to use minor order to represent an ordered set; however, the first instance has a NULL minor order. Is minor order used?
Action: Check for bogus entries. Determine if major order should be used to indicate side and if major order should be used to indicate an ordered set.

3.3.8 CosmidXCosWell

Number of instances:	415
Status:	Inconsistent use of major and minor order.
Purpose:	Indicates the cosmid in a well (of a microtitre dish). This relationship is currently being created by either Rob or Dave during cosmid library generation.



Relationship:	This relates a gvo Cosmid to an lmo Cosmid Well. This is a one-to-one relationship.
Major Order:	Null on first instance, Indicates "side" on later instances. Side 1 = Cosmid, Side 2 = Cosmid Well. The domain type for Cosmid is incorrectly recorded as "lmo" on earlier instances (should be domain type "gvo").
Order numbers:	Looks like minor order was used to indicate "side" on early instances. Current instances don't appear to use minor order; therefore, minor order should be NULL.
Action:	Correct use on major and minor order numbers. Correct lmrType from "lmo" to "gvo" on earlier instances.

3.3.9 CosmidxMtd

Number of instances:	52
Status:	Inconsistent use of minor order.
Purpose:	?
Relationship:	This relates gvo Cosmid to lmo Microtitre Dishes in a one-to-one relationship.
Major Order:	Not used.
Minor Order:	Not used in early instances, indicates "side" on current instances.
Action:	Determine if this can be deleted, if so delete it. If not, fix order numbers. Fix the bogus lowercase "x".

3.3.10 CosmidxProbe

Number of instances:	241
Status:	Consistent definition.
Purpose:	?
Relationship:	Relates gvo Cosmid to lmo Cosmid Probe. This is appears to be a one to many relationship.
Major Order:	Indicates "side". Side 1 = Cosmid, Side 2 = Cosmid Probe.
Minor Order:	Side 1 always has a minor order of 1, Side 2 appears to be used to order Cosmid probes.
Action:	Determine purpose. Determine if minor order is used.

3.3.11 DNA Template Mtd Set

Number of instances:	371
Status:	Inconsistent use of major and minor order.
Purpose:	?
Relationship:	Groups Individuals. Is this an ordered set?
Major Order:	Not used in early instances. Why are current instances using major order of "1" if this is a 1 sided relationship?
Minor Order:	Appears to be used for ordering the elements.
Action:	Determine purpose. Resolve use of major and minor order.

3.3.12 DNA Tempate Set

Number of instances:	418
Status:	Inconsistent use of major and minor order. Different member types.
Purpose:	Set of plates?
Relationship:	In an early instances, groups DNA Template Mtd Sets together. In current instances, groups Cosmid Mtds.

Major Order:	In early instances, major order is NULL. In current instances, major order appears to order the elements.
Minor Order:	In early instances, minor order appears to order the elements. In current instances, the minor order is not used.
Action:	Determine why meaning of major and minor order have been flip-flopped. Why do different rel member types participate in earlier instances of this relationship type? Fix use of major and minor orders and make rel member types consistent.

3.3.13 Fluorescent Image Set

Number of instances:	192
Status:	No discernable types in early instances.
Purpose:	?
Relationship:	Appears to group fluroescent images in an ordered set.
Major Order:	In early instances, not used. In current instances, appears to be used to order the elements.
Minor Order:	In early instances, appears to have been used to order elements. In current instances, minor order not used.
Action:	Determine purpose. Determine rel member types of early instances. Determine why major and minor order appear to have flip-flopped and fix use of order numbers.

3.3.14 Fluorescent Primer Set

Number of instances:	1
Status:	Only 1 instance.
Purpose:	?
Relationship:	Appears to group fluroescent primers in an ordered set.
Major Order:	Not used. Should be null.
Minor Order:	Appears to be used to order the elements.
Action:	Determine purpose. Determine use of major / minor order. Delete if not used.

3.3.15 ImgXCosFrg

Number of instances:	3814
Status:	Consistent definition.
Purpose:	?
Relationship:	Appears to relate an Image to a Cosmid Fragment in a one-to-one relationship.
Major Order:	Indicates "side". Side 1 = Image, Side 2 = Cosmid Fragment.
Minor Order:	Not used (NULL).
Action:	Determine purpose.

3.3.16 ImgXImgBand

Number of instances:	1335
Status:	Unable to discern rel member types that participate in this relationship.
Purpose:	?
Relationship:	Appears to relate Image to Image Bands in a one-to-many relationship.
Major Order:	Appears to indicate "side". Side 1 = Image, Side 2 = Image Bands.
Minor Order:	Not used (NULL).
Action:	Determine purpose. Correct rel member instances to appear in "objects" table. Determine if relationship has a consistently used definition.

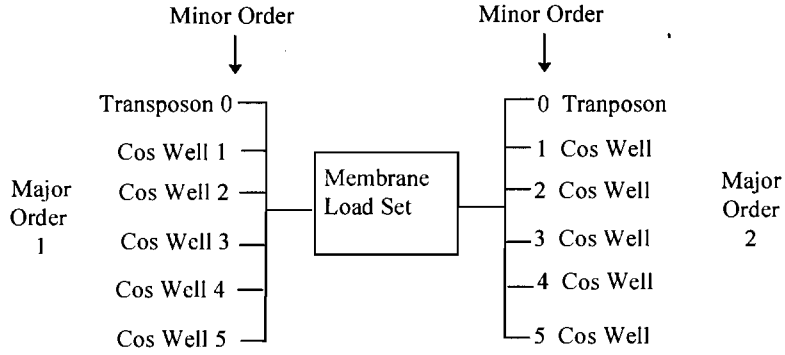
3.3.17 ImgXImgLane

Number of instances:	13835
Status:	Unable to discern rel member types that participate in this relationship.
Purpose:	?
Relationship:	Appears to relate Image to Image Lane in a one-to-one relationship.

Major Order: In early instances, appears to indicate "side" with Side 1 = Image and Side 2 = Image Lane. In last instance, the major order appears to be NULL. Is is a bug?
Minor Order: Not used in early instances; appears to indicate order of elements in last instance.
Action: Determine purpose. Correct rel member instances to appear in "objects" table. Determine if relationship has a consistently used definition.

3.3.18 Membrane Load Set

Number of instances: 44
Status: Consistent definition (lmrType = "dmo" for this relationship definition).
Purpose: Defines the layout of a multiplexed sequence membrane.
Relationship: Groups multiplex layers of a membrane. For each multiplex layer, relates a Cosmid Probe to Cosmid Wells in a one-to-many relationship.



Major Order: Indicates "side". (Multiplex layer of a membrane)
Minor Order: 0 = Transposon, 1-n = Cosmid Wells. 1-n represents an ordered set of Cosmid wells.
Action: Redesign relationship to not depend on 0 minor order to represent transposon and clarify notion of "sides".

3.3.19 STSXPrimer

Number of instances: 2320
Status: Inconsistent rel member types participate in this relationship.
Purpose: ?
Relationship: Appears to relate STS to Primer in a one-to-many relationship.
Major Order: Indicates "side". Side 1 = STS, Side 2 = Primer.
Minor Order: Used in early instances, not used in last instance.
Action: Determine purpose. Determine if last instance, which only has Primer, is correct or is an incomplete relationship. Fix database to make major and minor order consistent. Fix use of rel member types.

3.3.20 STSxInformPed

Number of instances: 122
Status: Consistent definition.
Purpose: ?
Relationship: Appears to relate STS to Pedigree in a many-to-many relationship.
Major Order: Appears to indicate "side". Side 1 = STS, Side 2 = Pedigree.
Minor Order: Is minor order "overloaded" such that Side 1, Minor Order 3 has a special meaning?
Action: Determine purpose. Clarify use of minor order.

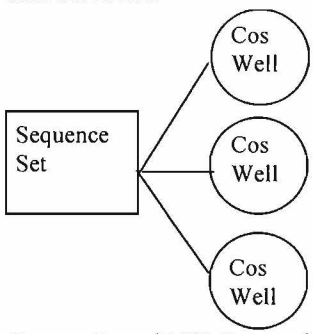
3.3.21 STSxYACPool92

Number of instances: 1357
Status: Consistent definition.
Purpose: ?
Relationship: Appears to relate YACPool92 to STS in a one-to-one relationship.

Major Order: Not used (null).
Minor Order: Not used (null).
Action: Determine purpose.

3.3.22 Sequence Set

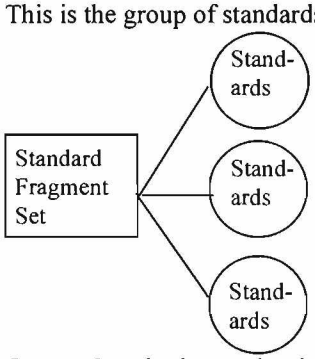
Number of instances: 167
Status: Consistent definition.
Purpose: This represents a minimal span set, which is the set of wells (from various Cosmid Mtd) that represent complete coverage of a Cosmid. TMAP creates this relationship. It is an unordered set.



Relationship: Groups Cosmid Wells together in an unordered set.
Major Order: Not used (null).
Minor Order: Not used (null).
Action:

3.3.23 Standard Fragment Set

Number of instances: 3
Status: Consistent definition.
Purpose: This is the group of standards that allows image bands to be properly sized in TMAP.



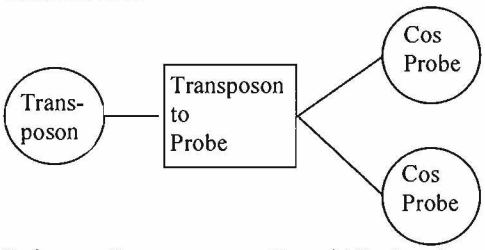
Relationship: Groups Standards together in an unordered set.
Major Order: Not used (null).
Minor Order: Not used (null).
Action:

3.3.24 StdFragXImgBand

Number of instances: 852
Status: Consistent definition.
Purpose: ?
Relationship: Appears to relate an Image Band to a Standard in a one-to-one relationship.
Major Order: Indicates "side". Side 1 = Image Bands, Size 2 = Standard.
Minor Order: Not used (null).
Action: Determine purpose.

3.3.25 TransposonXProbe

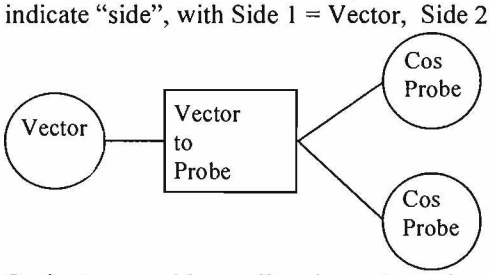
Number of instances: 8
Status: Consistent definition.
Purpose: Indicates which Cosmid Probes (K and S) are attached to a transposon. These probes allow the sequence to be read from the left and right side of a transposon which is attached to an insert. This relationship is create by Rob or Dave during library construction.



Relationship: Relates a Transposon to Cosmid Probes in a one-to-two relationship.
Major Order: Indicates "side". Side 1 = Transposon, Side 2 = Cosmid Probe.
Minor Order: Appears to indicate order of Cosmid Probes.
Action: Determine if relationship is one-to-two or one-to-many. Determine if minor order is used.

3.3.26 VectorXProbe

Number of instances: 47
Status: Inconsistent use of major order.
Purpose: Indicates the left and right side probes attached to a vector.
Relationship: Relates a Vector to Cosmid Probes in a one-to-two relationship.
Major Order: Some instances have a null major order, other instances appear to use major order to indicate "side", with Side 1 = Vector, Side 2 = Cosmid Probe.



Minor Order: On instances with a null major order, minor order appears to indicate "side". On other instances that use major order to indicate "side", minor order appears to order the Cosmid probes.
Action: Unravel the meaning of major and minor order and correct instances.

3.4 Empty Relationships

The following relationships exist in the database, but contain no members. The consensus is that they should be deleted:

CosFragXImgBand
CosLibXCosMtd
CosMtdXCosPool
CosMtdxCosFrg
CosTranxImage
CosXVector
CosmidXCosPool
DNATemplateMtdXPCRProdMtd
STSxHybridPanel
Sequence Image Lanes
YACxYACLib

4. Current Processes

4.1 Database Inconsistencies

4.1.1 Physical flow does not correspond to process inputs and outputs. This is due to a number of factors:

Applications are spanning a large section of the flow and thus represent an aggregate view of the flow.

Process inputs and outputs have evolved to handle changes to the flow.

Certain sections of the flow are not currently covered by applications.

In interest of deploying applications as quickly as possible, some applications do not address the process flow or address in it an isolated fashion.

4.1.2 Registered inputs do not correspond to the actual objects / relationships read from an application.

An example of this is the Membrane Layout Process. The registered input of this process in a Membrane Layout Definition. However, what is read by this application and processed as input is the relationship Sequence Set and the related Cosmid Mtds.

4.2 Definition of Current Processes

4.2.1 2 point localization

Number of instances:	1,492
Purpose:	?
Application:	?
Process Inputs:	gar Pedigree (many), gvo STS (1)
Process Outputs:	gvr ChrmXSTS (1)
Action:	Determine purpose and identify application.

4.2.2 BioMek Robot

Number of instances:	109
Purpose:	Retrieves the contents of specific Cosmid Mtd Wells which represent the minimal span set for a particular Cosmid?
Application:	?
Process Inputs:	lmr DNA Template Set (1), lmr STSxInformPed (1)
Process Outputs:	lmr DNA Template Mtd Sets (2), lmr DNA Template Set, lmo Mtd (2)
Action:	Clarify its purpose. Appears to not be used anymore (last process instance was created on Jul 27, 1994). Determine why it is no longer used.

4.2.3 Blot

Number of instances:	299
Purpose:	Transfers the contents of the gel onto the membrane via the direct blotter.
Application:	Mux screen.
Process Inputs:	lmo Gel (1).
Process Outputs:	lmo Membrane (1).
Action:	This process appears to have an overloaded meaning. It is also used to write the inputs and outputs of the Membrane Layout screen. Should these

processes be different? Does one represent blotting during TMAP and one during sequencing?

4.2.4 Chromosome Assignment

Number of instances: 0
Action: Delete.

4.2.5 Cube Multiplex

Number of instances: 322
Purpose: Combine contents of Cosmid Microtitre dishes to create multiplex layers for transposon mapping.
Application: Mux screen.
Process Inputs: lmo Cosmid Mtd (many).
Process Outputs: lmo Cosmid Pool Set (1), lmr CosPoolSetXCosMtd (many).
Action:

4.2.6 EPH Gel

Number of instances: 2480
Purpose: Inject contents from Cosmid Pool set along with Standard Fragment Set into the lanes of gel. Perform Electrophoresis to distribute the Cosmids along the lane of the gel according to length.
Application: Mux screen.
Process Inputs: lmo Cos Pool Set (1), lmr Standard Fragment Set (1).
Process Outputs: lmo Gel (1).
Action:

4.2.7 Fluorescent Genotyping

Number of instances: 2480
Purpose: ?
Application: ?
Process Inputs: gar Pedigree (1), gvo STS (1).
Process Outputs: lmo Isotopic Image (1).
Action: Determine purpose and identify application.

4.2.8 Gene Scan

Number of instances: 368
Purpose: ?
Application: ?
Process Inputs: Mtd (1), Standard Fragment Set (1).
Process Outputs: Fluorescent Image (1), Fluorescent Image Set (1).
Action: Determine purpose and identify application.

4.2.9 Hybrid Panel Test

Number of instances: 0
Action: Delete.

4.2.10 Informative Picks

Number of instances: 121
Purpose: ?
Application: ?
Process Inputs: STS (3).
Process Outputs: STSInformPed (1).
Action: Determine purpose and identify application.

4.2.11 Library Reformat

Number of instances: 0
Action: Delete.

4.2.12 Mate

Number of instances: 423
Purpose: Mate a Cosmid and Transposon.
Application: Mate screen.
Process Inputs: gvo Cosmid (1), lmo Transposon (1).
Process Outputs: lmr CosXTran (1).
Action:

4.2.13 Min Span

Number of instances: 174
Purpose: Find minimal span set that represents coverage of a Cosmid. This process is referred to as Transposon Mapping.
Application: TMAP.
Process Inputs: Cosmid (1).
Process Outputs: Seqence Set (1). (This represents the minimal span set).
Action: Try to link up to other part of the flow. This can be accomplished by linking up to the output from Probe. Since image splitting and image sizing are covered in applications, these processes could be added to the flow, feeding the output from probe to image splitting, image splitting to image sizing, image sizing to TMAP (min span).

4.2.14 Mtd Overlay

Number of instances: 212
Purpose: ?
Application: ?
Process Inputs: lmo Mtd (many).
Process Outputs: lmo Mtd (1).
Action: Determine purpose and identify application.

4.2.15 PCR

Number of instances: 975
Purpose: ?
Application: ?
Process Inputs: lmr DNA Template Mtd Set (1), lmo Fluorescent Primer (1).
Process Outputs: lmo Mtd (1).
Action: Determine purpose and identify application.

4.2.16 Pick

Number of instances: 329
Purpose: Colony Pick. Retreive mated Cosmids / Transposons from petrie dishes and place in Microtitre dish wells.
Application: Mux screen.
Process Inputs: lmr CosXTran (1).
Process Outputs: lmr TransTransXCosMtd (many), lmo Cosmid Mtd (many).
Action:

4.2.17 Polymorphism test

Number of instances: 0
Action: Delete.

4.2.18 Primer Coupling

Number of instances:	450
Purpose:	?
Application:	?
Process Inputs:	gvo STS (1).
Process Outputs:	lmo Fluorescent Primer (1).
Action:	Determine purpose and identify application.

4.2.19 Probe

Number of instances:	2,727
Purpose:	Expose a set of membranes to a probe and fluorogenic substrate to light up the position of the DNA fragment on the membrane. A CCD camera records an image which represents a set of the membranes that were placed in the drum of the probe chamber. This process actually represents a probe cycle, which will expose one of the multiplexed layers, based on the probe, of the each membrane loaded into the probe chamber. Note: The probe chamber is used for both Transposon mapping and Sequencing.
Application:	Probe Chamber -- or some other app?
Process Inputs:	lmo Cosmid Probe (1), lmo Membrane (1).
Process Outputs:	lmo Image (1).
Action:	Devise a better way of representing the inputs and outputs. For example, inputs could be multiple membranes, rather than 1 membrane and multiple probes rather than one probe. Outputs could be a meta-image, rather than a single image. Rename this process to Probe Chamber Cycle and Clean up old instances of this process that have different input and output types. Have probe chamber application write out process inputs and outputs.

4.2.20 STS Salt Trial

Number of instances:	0
Action:	Delete.

4.2.21 STS to LabPrimer

Number of instances:	0
Action:	Delete.

4.2.22 STSProbeYACPool28

Number of instances:	49
Purpose:	?
Application:	?
Process Inputs:	gvo STS (1), lmo YACPool92 (1).
Process Outputs:	gvr STSxYac (1).
Action:	Determine purpose and identify application.

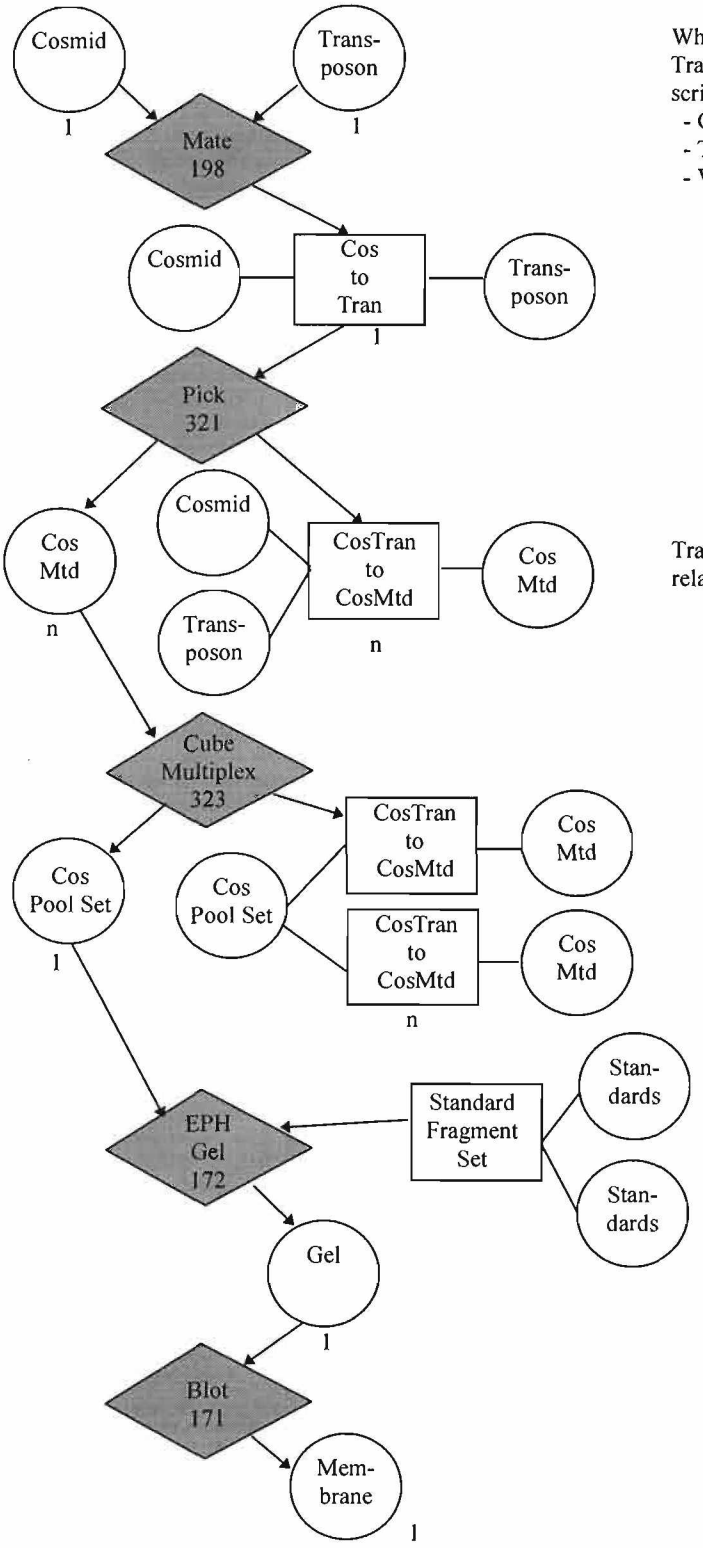
4.2.23 STSProbeYACPool92

Number of instances:	166
Purpose:	?
Application:	?
Process Inputs:	gvo STS (1).
Process Outputs:	lmo YACPool92 (many), lmr STSxYAXPool92 (many).
Action:	Determine purpose and identify application.

4.2.24 Sequence Run Selection

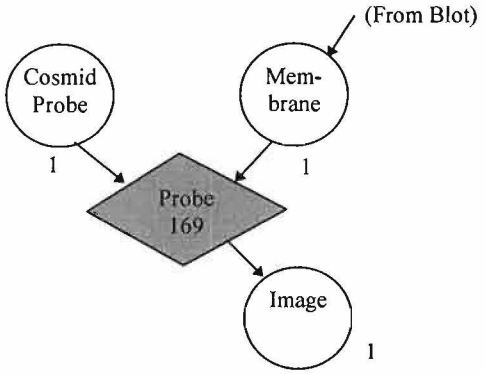
Number of instances:	0
Purpose:	?
Application:	?
Process Inputs:	
Process Outputs:	
Action:	Delete.

4.3 Current Process Flow



What relationships for Cosmids and Transposons were created via adhoc sql scripts during library construction?
- CosmidXCosWell?
- TransposonXProbe?
- VectorXProbe?

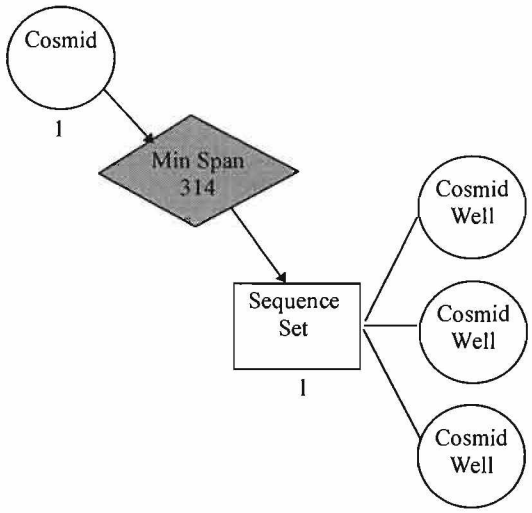
Tran-to-CosMtd should be a one-to-many relationship.



Can cosmid probe be determined by CosmidXProbe relationship? Is a particular membrane associate with a Cosmid by walking the process flow back to "Cube Multiplex" input of Cos MTD? Should a relationship be created to prevent such a long traversal?

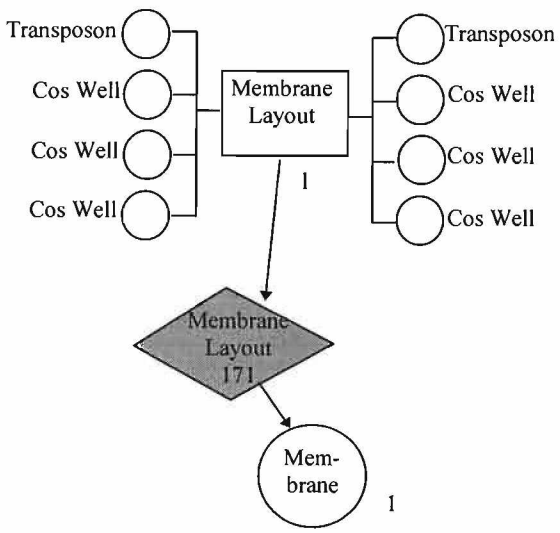
Devise a better representation of the Probe process to record that multiple probes and membranes can go into a probe cycle and the output from a probe cycle is actually an image of multiple membranes.

Should "tv" application write process inputs and outputs to represent splitting of images from probe chamber and relating individual images to its content (i.e. Cosmid)



What inputs should feed Min Span to "link up" the outputs from Probe?

Min Span is defined as a Method in the database; this should be a "lmp" process.



Devise a different way of representing the inputs to the Membrane Layout process so that:

1) The input the sequence set generated from TMAP (Min Span process).

2) The output represents a membrane definition, describing the multiplex layers and their contents.

3) The relationship between transposon and Cos Well is expressed using "sides" of a relationship rather than minor order.

Why does the membrane layout screen record its process using "Blot"? Isn't this a different process in the flow? Are we trying to represent the same processes that go on during TMAP?

Can any of the seqence processes currently being recorded to the database be recorded as processes with inputs and outputs?

5.