Technical Report NEESgrid-2003-21
www.neesgrid.org

(Draft Whitepaper Version: 1.0
Last modified: October 4, 2003)

NMDS Reference User Guide

(DRAFT)

Joe Futrelle[1]

[1] National Center for Supercomputing Applications, Champaign, IL 61821

Feedback on this document should be directed to futrelle@ncsa.uiuc.edu

Acknowledgment: This work was supported primarily by the George E. Brown, Jr. Network for Earthquake Engineering Simulation (NEES) Program of the National Science Foundation under Award Number CMS-0117853.

1 Summary

The NEESgrid metadata service (NMDS) API provides a means for creating, managing, retrieving, and modifying metadata objects in a NEESgrid repository. This document describes the NEESgrid repository object model, gives an overview of the NMDS API, and demonstrates each major capability of the NMDS API with code examples. This document describes the NMDS API as of NEESgrid release 2.0.

1.1 About this document

This document assumes some familiarity with the NEESgrid repository project. Readers of this document should review earlier documents, including:

- NEESgrid Technical Report 2003-15: NEESML Reference User Guide
- NEESgrid Technical Report 2003-12: Overview of NEESML Whitepaper v1.0
- "Designing Metadata for the NEESgrid Data Repository" http://www.ncsa.uiuc.edu/People/futrelle/ppt/training_metadata.ppt (April 2003)
- NEESgrid Technical Report 2002-05: The NEESgrid Metadata Service API: Overview
- NEESgrid Technical Report 2002-04: NEESgrid Data and Metadata Harvesting Protocol Whitepaper v1.0.2

Since the NEESgrid repository implementation is still under development, some older documents may be out of date. Newer documents take precedence. This document describes the NMDS API as of NEESgrid release 2.0, and can be referred to when developing applications against that release.

In addition, it can be useful to review other relevant technologies. A high-level understanding of XML, XML Schema, and RDF will help contextualize the framework being described here.

This document describes the NEESgrid repository object model, gives an overview of the NMDS API, and demonstrates each major capability of the NMDS API with code examples.

Many of the code examples are abstract, rather than taken from applications in any specific domain. This is done in the interest of brevity.

2 NEESgrid Repository Object Model

2.1 Overview

NEESgrid metadata objects are based on a simple, RDF-like model. In the model, each object can be associated with primitive values and to other objects by means of named relations. In addition, object types can be defined which constrain the kinds of relations that are permitted for each object of that type. Using relations, web-like topologies of

linked objects can be created that can be used to describe real-world or application-specific relationships.

Objects are immutable. This means that the relations of an object cannot be modified. Instead, new versions of objects can be created. This simplifies concurrency control but complicates the API considerably. It is important to make sure that you pay attention to issues of versioning and concurrency when you write clients, which may interact with other clients accessing the same objects simultaneously.

NMDS also provides a set of convenience API's for adding and removing objects from containers. Containers are special objects that can contain other objects, much the same way a folder contains files in Windows. Like folders, containers can contain other containers. Unlike folders and files in Windows, objects can be in more than one container simultaneously.

2.2 Important concepts

2.2.1 Objects

The "unit of currency" in the NEESgrid repository is the object. Objects, also called instances, encapsulate sets of related data items. Objects can be used either to represent real-world objects, such as a beam or a sensor, or abstractions, such as an experimental design or a schedule. They accomplish this representation using relations. An object is analogous to, but not equivalent to, a row in a spreadsheet or relational database table or a resource in RDF.

2.2.2 Relations

Relations are named data items associated with an object. An object is little more than a collection of relations. Each relation has a name and an associated data item, called its value, which is either a primitive data item such as an integer or a date, or a link to another object. A relation is analogous to, but not equivalent, to a cell in a spreadsheet or column value in a relational database table. A relation may have any number of values for a single object.

2.2.3 Types

A type constrains the relations that an object can contain, both in terms of type and cardinality. Objects can be generated which conform to types, and objects can be compared to types to check their validity. Types are analogous to, but not equivalent to, a table definition in a relational database or a class in Java or C++.

2.2.4 References

References allow objects to be unambiguously identified. A reference identifies an object using a unique identifier and a version number. Relations can have references as values, forming a link between objects. A reference is analogous to, but not equivalent to, a foreign key in a relational database table or an IDREF in an XML document.

2.2.5 Versioning

When an object is initially created, its version number is 0. Every time the object is updated, a new object with a higher version number is created. Clients can use version numbers to reconstruct the web of objects that existed in the repository at any point. They can also use version numbers to make decisions about how to handle contention with other clients over updating the same objects. A client can always get a reference to the latest version of any object.

2.2.6 Containers

Containers are objects that can contain other objects. NMDS provides a number of useful methods for working with containers, including traversing container hierarchies, listing the contents of containers, adding and removing objects from containers, and searching in containers. A container is analogous to, but not identical to, a folder in Windows or a Set in Java.

3 Overview of the NMDS API

The NMDS API acts as a communication layer between an application and the NEESgrid repository. Because NMDS is based on web services, the application need not be co-located with the repository. API calls that require exchanging information with the repository are automatically translated into web services requests by the API, which manages the connection to the repository.

The NMDS API consists of two sets of interfaces and some utility classes. The first set of interfaces represents objects, relations, and object references. The second set of interfaces represents the metadata service. And finally, utility classes provide simplified API's for constructing and managing types, testing the service, ingesting XML files containing object descriptions, and other useful functions.

3.1 Object interfaces

`org.nees.md.MetadataObject`

This interface represents a NEESgrid metadata object. This is the interface that lets you retrieve and modify the relation values for an object. It is in effect an in-memory representation of one version of a metadata object, which can be used both to retrieve relation values and modify relation values so that the modified object can be uploaded to the repository as a new version. Because links between objects are represented as relations, this is also the interface that makes it possible to traverse the links between objects.

`org.nees.md.Reference`

This interface represents a reference to a NEESgrid metadata object. It identifies both the object and version number, making it possible to retrieve any version of a given object. References can be used as the values of relations; see below.

`org.nees.md.Relation`

This interface represents a relation that associates an object with a value. The value may be either of a primitive type or a reference. Each relation is named with an identifier, which distinguishes it from other relations on the same object or in the same type. You will rarely use this interface directly; instead, convenience methods are provided in the NMDS API that allow relation values to be accessed and modified directly.

3.2 Service interfaces

`org.nees.repo.service.RepositoryService`

This interface provides access to a NEESgrid repository. It allows you to create, retrieve, and update objects, as well as to retrieve information about objects. It also allows you to add and remove objects from containers, and to search for objects according to the values of their relations. This interface extends `org.nees.md.service.MetadataService`, a simpler service that provides basic functionality.

3.3 Utility classes

`org.nees.md.Namespace`

This class provides methods for constructing and parsing ID's with namespace parts.

`org.nees.md.MetadataFactory`

This class provides a safe means of constructing objects implementing the `MetadataObject`, `Reference`, and `Relation` interfaces. Safe, meaning it will generate instances of whatever class is most appropriate for the implementation of NMDS your application is using.

`org.nees.repo.Schema`

This utility class provides a number of methods for creating, modifying, and managing object types, as well as a variety of other utility methods.

4 Interacting with the repository

Meaningful interactions between applications and the repository follow a number of specific patterns, which are described below.

4.1 Stages of interaction

All meaningful NMDS interactions proceed through at least three of these stages, in order:

4.1.1 Acquiring a service instance

The first stage of any interaction with the repository is to acquire a service instance. This is always an instance of `org.nees.repo.service.RepositoryService`. This is done using `org.nees.repo.service.RepositoryServiceFactory` (see code example in section 5.1.2). The factory service instance must be configured so that it can locate the repository service. How it is configured depends on the factory implementation (see code example in section 5.1.1). Once an application has acquired a single service instance, it can perform any number of operations with the repository, subject only to the continuing availability of the repository (which for instance may become unavailable if the network goes down).

4.1.2 Acquiring a reference

In order to retrieve any object from the repository, the client must have a reference to some version of that object. Applications can create references if they know the ID of an object in the repository, or they can create an object of a known type, which will return a valid reference. Every NEESgrid repository contains a "root container" whose ID is known and which contains all of the other objects visible to the user in the CHEF repository browser (see code example in section 5.1.3). In addition, a client can query the repository for objects that match a criterion, and acquire references that way (see section 5.6).

4.1.3 Retrieving an object

To find out the values of an object's relations, the object must be retrieved. This can be done either explicitly (see code example in section 5.2.1), or implicitly, by querying the service for the values of individual relations (see code example in section 5.2.6). Once a client has retrieved an object, it can use the object indefinitely, even if the repository becomes unavailable. The object will never expire, although other clients may create new versions of it. Some of the values of an object's relations may be references, which the client can use to retrieve other objects (see code example in section 5.2.7). In this manner it can traverse the web of relations defined in the application or domain-specific metadata model.

4.1.4 Modifying an object

To modify an object, a client must create a new version of it. This is done by modifying a `MetadataObject` and submitting it to `RepositoryService` as a new version. This may fail, because the object may have been modified by another client since the first client retrieved it. In this case, NMDS throws an `org.nees.md.service.ConcurrentAccessException`, and the client must decide (with or without user input) whether to abandon its change, override the other client's change, or merge its changes to the object with the changes made by the other client. For code examples see section 5.3.3.

5 Code examples

The following code examples demonstrate typical NMDS interactions.

5.1 Basic interactions

5.1.1 Creating and registering a factory

To acquire a `RepositoryService` instance, you must first create a factory. If you register the factory using `RepositoryServiceFactory`'s `setFactory` method, you can then use it at any time afterwards to create an NMDS instance. The following example creates and registers a factory that connects to a service at `http://neespop.mysite.edu:8030/axis/services/nmds`.

```
import java.net.URL;
import org.nees.repo.service.RepositoryServiceFactory;
import org.nees.repo.axis.client.NMDSClientFactoryImpl;

…

URL url = new
URL("http://neespop.mysite.edu:8030/axis/services/nmds");

// create and register an NMDSClientFactory
RepositoryServiceFactory.setFactory
  (new NMDSClientFactoryImpl(url));
```

5.1.2 Acquiring a service instance

Once you have registered a factory, acquiring a service instance takes just one call to `RepositoryServiceFactory`'s `newRepositoryService` method.

```
import org.nees.repo.service.RepositoryService;

…

RepositoryService service =
RepositoryServiceFactory.getFactory().newRepositoryService(
);
```

5.1.3 Creating a reference

To create a reference, use the metadata factory's `newReference` method. You can create a reference to any version of the object. If you do not specify a version, the version will default to 0.

```
import org.nees.md.Reference;
import org.nees.md.MetadataFactory;

…

String id = "i483583742.43";
int version = 5;

Reference myReference =
MetadataFactory.getFactory().newReference(id, version);
```

5.2 Retrieving objects and relations

5.2.1 Retrieving objects

To retrieve an object, pass a reference to it to the repository service's `get` method. For instance, to get the object referred to in the code fragment in section 5.1.3, do this:

```
import org.nees.md.MetadataObject;
…
MetadataObject myObject = service.get(myReference);
```

If the reference refers to a nonexistent object, `get` throws an `ObjectNotFoundException`. To retrieve a group of objects, pass an array of `References` to `get`. Like this:

```
Reference[] someReferences;

… do something to populate someReferences …

MetadataObject myObjects[] = service.get(someReferences);
```

5.2.2 Retrieving different versions of an object

The repository may contain multiple versions of a given object. Each reference refers to a specific version of an object. Suppose you have an object called myObject. To get a reference to that object, call getSelf on the object:

```
Reference refToObject = myObject.getSelf();
```

To find out the object's version, call getVersion on the reference:

```
int version = theVersion = refToObject.getVersion();
```

To get a reference to the previous version of the object, create a reference with a decremented version number:

```
Reference previousVersion =
    MetadataFactory.getFactory().newReference
        (refToObject.getID(), refToObject.getVersion() -
1);
```

To get the first version of the object, create a reference with version number 0.

```
Reference firstVersion =
    MetadataFactory.getFactory().newReference
        (refToObject.getID(), 0);
```

To get a reference to the latest version of an object, call `RepositoryService`'s `getLatestVersion` method:

```
Reference latestVersion =
```

```
service.getLatestVersion(refToObject);
```

Note that this will return a reference to the latest version of the object at the time `getLatestVersion` is called. If the object is subsequently updated, the reference will not change.

5.2.3 Retrieving the root container

To retrieve the root container of a NEES repository, you must first create a reference to it, then retrieve it. The ID of the root container is "`nees:rootContainer`". Unless you know which version of the root container you want to retrieve, you should retrieve the latest version of it:

```
import org.nees.md.Reference;
import org.nees.md.MetadataFactory;

…

Reference ref = MetadataFactory.getFactory().newReference
  ("nees:rootContainer");

// get the latest version of the container
ref = service.getLatestVersion(ref);
MetadataObject rc = service.get(ref);
```

For examples of how to work with containers, see section 5.4.

5.2.4 Getting relation values

Once an object has been retrieved, the values of its relations become available. The following subsections demonstrate how to retrieve relation values from an object. They assume that an object has already been retrieved, and that it is stored in a variable called `myObject`. They further assume that the object has two relations called "`aRelation`" and "`anotherRelation`".

5.2.5 Getting all relations

To get all the relations of an object, call `MetadataObject`'s `getRelations` method. This is rarely what you will want to do; instead, you will most likely want to get selected relation values, described in section 5.2.6.

```
Relation[] theRelations = myObject.getRelations();
```

Each `Relation` has an ID and a target. To get the ID of a Relation, call Relation's getID method:

```
for(int j = 0; j < theRelations.length; j++) {
    String rID = theRelations[j].getID();
    … do something …
}
```

An object may have more than one `Relation` with the same ID. The targets associated with a given relation ID are considered the values of the relation on that ID for the given object. There are six types of targets: date, double, int, long, reference, and string. To find out a Relation's target type, call `Relation`'s `getType` method:

```
for(int j = 0; j < theRelations.length; j++) {
    String rID = theRelations[j].getID();
    String rType = theRelations[j].getType();
    … do something …
}
```

There are six target type ID's, one for each target type, that are defined as constants in the Relation class: `DATE_TYPE_ID`, `DOUBLE_TYPE_ID`, `INT_TYPE_ID`, `LONG_TYPE_ID`, `REFERENCE_TYPE_ID`, and `STRING_TYPE_ID`. To get the target of a relation, you can either call `getTarget`, or one of the convenience methods `getDateTarget`, `getDoubleTarget`, `getIntTarget`, `getLongTarget`, `getReferenceTarget`, and `getStringTarget`:

```
for(int j = 0; j < theRelations.length; j++) {
    String rID = theRelations[j].getID();
    String rType = theRelations[j].getType();
    if(rType.equals(Relation.INT_TYPE_ID)) {
        Integer k = (Integer) theRelations[j].getTarget();
        … or: …
        int k = theRelations[j].getIntTarget();
    }
}
```

If you only want to get the ID's of all the relations, you can call `getRelationIDs`. This will return only the set of distinct relation ID's for this object.

5.2.6 Getting selected relation values

Rather than calling `getRelations`, it is much more convenient to get selected relations whose ID's you know. This can be accomplished with `MetadataObject`'s methods `getDate`, `getDouble`, `getInt`, `getLong`, `getReference`, and `getString`.

```
String valueOfARelation = myObject.getString("aRelation");
Reference valueOfAnotherRelation =
    myObject.getReference("anotherRelation");
```

This will throw a `RelationTargetTypeException` if the value of the relation for the object is not of the expected type (e.g., if you called `getDate` on a relation whose value was of type string).

To get multiple values for a selected relations, call one of the following methods: `getDates`, `getDoubles`, `getInts`, `getLongs`, `getReferences`, and

`getStrings`. This is also the preferred technique to find out how many values a relation has.

```
String valuesOfARelation[] =
    myObject.getStrings("aRelation");
if(valuesOfARelation.length > 0) {
    … do something …
}
```

Relations are unordered sets, so you should not write code that depends on the order the values are returned in the array.

5.2.7 Following a link

Relations whose values are references allow a client to follow a link from one object to another. Suppose `myObject`'s `anotherRelation` relation has a reference to another object as its value. We can get the reference and use it to retrieve the other object:

```
Reference ref= myObject.getReference("another Relation");
MetadataObject otherObject = service.get(ref);
```

Now we can get the values of `otherObject`'s relations. Recursively applying this principle can allow us to use hierarchical object networks, as well as other, web-like networks of objects linked together by references.

5.2.8 Titles and creation/update times

There are several pieces of information you can find out about an object given a reference to it. You can find out the descriptive title of an object, as well as when the object was created and updated. To find the title, call `RepositoryService`'s `getTitle` method:

```
String title = service.getTitle(ref);
```

To find out when the object was created, call `getTimeCreated`:

```
java.util.Date creationTime = service.getTimeCreated(ref);
```

This is the time at which the first version of the object was created. To find out when the object was updated, call `getTimeUpdated`:

```
java.util.Date updateTime = service.getTimeUpdated(ref);
```

This is the time that the version of the object referred to by `ref` was created. Both of these times are represented with millisecond resolution.

5.3 Modifying the repository

Once an object is put in the repository, it cannot be modified. However, a new version of the object can be put in the repository.

5.3.1 Creating objects

To create an object of a given type, use `RepositoryService`'s `create` method. Pass it the ID of the type, and a descriptive title, and it will return a reference to the newly-created object. For more about types see section 5.5.

```
Reference aNewObject = service.create
    ("aTypeID", "my object");
```

If there is no type with the given type ID, `create` will throw a `TypeNotFoundException`.

If you know the ID you want to give the object, you can pass it in as a third argument to create. This ID must not be in use anywhere in the repository, or `create` will throw a `MetadataServiceException`.

5.3.2 Deleting objects

To delete an object, call `RepositoryService`'s `delete` method. This will create a new version of the object which is marked as deleted, and return a reference to the new version. You can check if an object has been deleted by calling `isDeleted`.

```
Reference deletedObject = service.delete(ref);
if(service.isDeleted(service.getLatestVersion(ref))) {
    … do something …
}
```

There is no way to remove existing object versions from the repository.

5.3.3 Updating objects

Updating an object—creating a new version of the object with modified relation values—is a three-step process. First, you must retrieve the object. Second, you must modify the relation values. And finally, you must create a new version of the object based on your modifications.

5.3.3.1 Modifying relation values

As with retrieving relation values, there are sets of six methods defined on MetadataObject for modifying relation values, one for each relation target type. To set a relation value, call set and pass it the relation ID and a value of the appropriate type:

```
myObject.set("aRelation", "a new value for the relation");
myObject.set("anotherRelation", 48.275);
```

`set` replaces any existing relation value with the one you pass in. To add a value to a relation, call `add`, similarly (continuing our example):

```
myObject.add("aRelation", "an additional value");
myObject.add("anotherRelation", -42.5);
```

In this case each relation will now have two values. To remove all values for a relation, call remove (continuing our example):

```
myObject.remove("aRelation");
```

In this case the relation with ID "aRelation" now has no values. To remove just one value, pass the value into remove as well (continuing our example):

```
myObject.remove
    ("aRelation", "a new value for the relation");
```

In this case the relation with the ID "aRelation" now has one value. The equals method is used for comparison. If there are two or more relation targets that match, only the first relation is removed. So in this example

```
anObject.add("temperatures", 74);
anObject.add("temperatures", 73);
anObject.add("temperatures", 74);
anObject.remove("temperatures", 74);
```

anObject has two relation values for "temperature" at the end, one of which is 74 and the other of which is 73.

5.3.3.2 Creating a new version of an object

Once you have modified all the relations you want to modify, pass the MetadataObject to RepositoryService's update method. This example retrieves an object, modifies one of its relations, and calls update:

```
MetadataObject person = service.get(refToPerson);
person.set("weight", person.getDouble("weight") - 3.2);
refToPerson = service.update(person);
```

update returns a reference to the new version of the object. You can only update the latest version of an object.

5.3.4 Handling concurrency issues

Multiple clients may access the repository simultaneously. Because of this, one client may update an object while another client is modifying it. When the second client tries to update the object, the version that it based its modifications on is no longer the latest version, and the update will fail with a ConcurrentAccessException. In fact, the only point at which the latest version of an object can be unambiguously determined is the point at which an update succeeds.

When writing a client, you must catch ConcurrentAccessException every time you call update, and take one of the following actions:

1. abandon the update

2. retrieve the latest version, merge it and your version, and retry the update

Because this choice, and the details of how to merge objects, will be different for different applications, the NMDS API will not automatically take either of these actions for you. In this example, the client abandons the update:

```
MetadataObject person = service.get(refToPerson);
person.set("weight", person.getDouble("weight") - 3.2);

try {
  refToPerson = service.update(person);
} catch(ConcurrentAccessException x) {
  // do nothing …
}
```

In this version, the client tries to subtract 3.2 from the value of the "`weight`" relation, even if the object is updated while the client is trying to make the modification:

```
while(true) {
    try {
        refToPerson =
service.getLatestVersion(refToPerson);
        MetadataObject person = service.get(refToPerson);
        person.set
            ("weight", person.getDouble("weight") - 3.2);
       refToPerson = service.update(person);
    } catch(ConcurrentAccessException x) {
        continue;
    }
    break;
}
```

5.3.5 About versioning and links

A link (a reference from one object to another) links an object to a specific version of another object. If the linked-to object is updated, the link will still refer to the older version of the object. In other words, if object α_i (i.e., object α, version i) links to object β_j, and object β is updated (creating object β_{j+1}), α_i will still link to β_j, not to β_{j+1}. For some situations and applications, this may be the correct behavior. If it's not, your application will need to update α when it updates β. An alternative strategy is: after following a link, retrieve the latest version of the linked-to object, not the version that is specifically linked to.

5.3.6 Copying objects

To copy an object, pass a reference to that object to `RepositoryService`'s `copy` method. This will return a reference to a new object, which is identical in every way except that it has a different ID, and a version number of 0.

```
Reference aCopy = service.copy(ref);
```

Copy only copies the object's relation values. It does not copy any objects that may be linked to the object by relations whose target type is reference (see section 5.2.7).

5.4 Containers

The NMDS API provides a means of organizing objects in containers. Containers are objects that, like all other objects, can link to sets of other objects. However, containers provide convenient API's for managing their contents.

5.4.1 Creating a container

To create a container, call `RepositoryService`'s `createContainer` method. Pass it a descriptive title. It will return a reference to the new container. Containers are initially empty.

```
Reference myContainer =
    service.createContainer("My new container");
```

5.4.2 Getting the contents of a container

To get the contents of a container, call `RepositoryService`'s `getContents` method on a reference to the container. It will return an array of references to the objects in the container. There is no guarantee that the objects will be in any particular order in the array, or that the order will be the same if you call this method more than once on a given container.

```
Reference cr; // a reference to a container
…
Reference contents[] = service.getContents(cr);
```

If any of the contents of a container are themselves containers, you can get their contents using this same method.

5.4.3 Determining if an object is a container and traversing a container hierarchy

To determine if an object is a container, call `RepositoryService`'s `isContainer` method on a reference. You can combine this with `getContents` to traverse a container hierarchy, as in this example:

```
void traverse(Reference c) {
    Reference contents[] = service.getContents(c);
    for(int j = 0; j < contents.length; j++) {
        if(service.isContainer(contents[j]) {
            traverse(contents[j]);
        } else {
            … do something with each object …
        }
    }
}
```

There is no restriction on which objects can be in which containers. A container could even be in itself. If a container network contains loops (e.g., a container contained in itself, or a container α containing container β which itself contains container α), a simple traversal algorithm such as the one above will never terminate. Either avoid creating loops in your container network, or use a different traversal algorithm.

5.4.4 Determining what containers an object is in

To find all the containers that an object is in, you can call `RepositoryService`'s `getCurrentContainers` method, but beware: this will return any version of any container that contains the object, so it might return more than one version of each container, and it might return containers which once contained the object, but which no longer contain it. If you are just interested in which containers currently contain the object, call `getCurrentContainers`, like this:

```
Reference myObject;
…
Reference containers[] =
service.getCurrentContainers(myObject);
if(containers.length == 0) {
    System.out.println("object " + myObject + " is not in
any container");
}
```

This call can sometimes be slow, especially if the object has ever been in a container that was updated many times.

5.4.5 Adding an object to a container

To add an object to a container, use `RepositoryService`'s `add` method. This will create a new version of the container that contains the object you added, and return a reference to the new version of the container. *Retain this reference unless you specifically want to use the older version of the container*. This is correct:

```
Reference myContainer, objectToAdd;
…
myContainer = service.add(objectToAdd, myContainer);
```

This is incorrect in almost every situation:

```
Reference myContainer, objectToAdd;
…
service.add(objectToAdd, myContainer); // WRONG!!
```

Since `add` modifies the repository, you must catch `ConcurrentAccessException` when you call it.

5.4.6 Removing an object from a container

Removing an object from a container is done with the `remove` method. The syntax is exactly the same as `add`, except that it has the effect of creating a new version of the container that no longer contains the object you removed.

Removing an object from a container does not delete the object. Indeed, after you remove an object from a container it may still be in any number of other containers from which you did not delete it. Deleting objects and removing them containers must be done separately. If you delete an object but do not remove it from a container, the container will still contain the version of the object immediately preceding the deleted version. To make an object completely inaccessible, remove it from all containers that it is currently in, and then delete it:

```
Reference objectToDelete;
…
service.removeLatestFromCurrentContainers
  (objectToDelete, true);
service.delete(objectToDelete);
// you must catch ConcurrentAccessException here
```

See section 5.4.8 for more details on `removeLatestFromCurrentContainers`.

5.4.7 Concurrency

If you want to add or remove an object from a container regardless of whether or not other clients have modified it since you acquired a reference to it, use the `addToLatestVersion` and `removeFromLatestVersion` variants of `add` and `remove`. These routines acquire a reference to the latest version of the container and update it. This is different from calling `getLatestVersion` on the container and then calling `add` or `remove` on it, because if you do that, another client can modify the container between the call to `getLatestVersion` and the call to `add` or `remove`. See section 5.4.8 for code examples.

5.4.8 Updating container hierarchies

When an object is updated, it is often desirable to update any containers it is in as well as any containers above them in the container. If this is not done, it may not be possible to access the modified object by traversing the container hierarchy. To update containers when updating an object, use `RepositoryService`'s `updateAndUpdateContainers` method in place of `update`:

```
MetadataObject person = service.get(refToPerson);
person.set("weight", person.getDouble("weight") - 3.2);
refToPerson = service.updateAndUpdateContainers(person);
```

Because adding and removing an object from a container updates the container, there are variants of `add` and `remove` that update containers. `addAndUpdateContainers` works like `add`, except it updates all the ancestor containers of the container to which you are adding an object; `removeAndUpdateContainers` works the same way for

removing an object. `addToLatestVersion` and `removeFromLatestVersion` provide a flag specifying whether you want to update containers. For instance, to remove an object from a container and update all the containers in the hierarchy, use this construct:

```
Reference myObject, myContainer;
…
myContainer =
  service.removeFromLatestVersion
    (myObject, myContainer, true);
```

To completely remove an object from all the containers that currently contain it, and update the container hierarchy, use the `removeLatestFromCurrentContainers` method. It will return an array of references to all the containers that contained the object, and were updated.

```
Reference myObject;
…
service.removeLatestFromCurrentContainers(myObject, true);
```

5.5 Types

Every object has a type. Types can extend other types in a hierarchy of types, much the way classes extend other classes in Java. Types constrain the allowed relations for an object, as well as the target type and cardinality of each relation. The NMDS API provides a number of convenience methods for determining type membership and finding out relation constraints. The `org.nees.repo.Schema` class provides a comprehensive set of methods for creating, modifying, and querying types.

5.5.1 Getting the type of an object

Each type is identified with a unique ID. To get the ID of the type of an object, call `RepositoryService`'s `getType` method, passing it either a reference to the object or the object itself.

```
MetadataObject myObject;
…
String typeID = service.getType(myObject);
```

or

```
Reference myReference;
…
String typeID = service.getType(myReference);
```

5.5.2 Determining type membership of an object

Suppose type α extends type β, which itself extends type χ. Every object of type β is considered to be of type χ, and similarly, every object of type α is considered to be of type β and therefore of type χ. There is a "root type" of which all types are extensions, either directly or through a chain of extensions.

To determine the type membership of an object, use `RepositoryService`'s `isa` method, passing it a reference to the object and the ID of the type you want to test. Suppose we have two types representing two different kinds of sensors, with ID's "`sensor:lvdt`" and "`sensor:sg`". Further suppose that each of those types extends a type with the ID "`sensor:sensor`". The following code determines if a reference refers to a sensor, and if so, what kind:

```
Reference aSensor;
…
if(service.isa(aSensor, "sensor:sensor")) {
    if(service.isa(aSensor, "sensor:lvdt")) {
        System.out.println("sensor is an LVDT");
    } else if(service.isa(aSensor, "sensor:sg")) {
        System.out.println("sensor is a strain gauge");
    }
} else {
    System.out.println("object is not a sensor");
}
```

5.5.3 Relation constraints

An object's type constrains which relations it may have, as well as the target type and cardinality of each relation[1]. To find out which relations are allowed for a type, use `Schema`'s `getRelations` method, passing it the type ID. It will return an array of relation ID's.

```
RepositoryService service;
String typeID;
…
String[] relations =
    (new Schema(service)).getRelations(typeID);
for(int j = 0; j < relations.length; j++) {
    System.out.println
        (relations[j] +
         " is defined for type with ID " +
         typeID);
}
```

To find out the target type of a relation, use `RepositoryService`'s `getRelationType` method and pass it the type ID and relation ID. It will return one of the target type ID constants from the `Relation` interface (see section 5.2.5 for a list).

```
String relationID;
…
String myObjectsType = service.getType(myObject);
String rType = service.getRelationType
    (myObjectsType, relationID);
```

[1] As of release 2.0, NMDS does not enforce relation constraints. Validation against relation constraints is planned for later releases.

```
if(rType.equals(Relation.DOUBLE_TYPE_ID)) {
    double values[] =
        myObject.getDoubles(relationID);
}
```

To find out the cardinality of a relation, use `RepositoryService`'s `getRelationMin` and `getRelationMax` methods, which return the minimum and maximum number of values allowed for a given relation on a given type. If there is no maximum, `getRelationMax` will return `Schema.CARDINALITY_UNBOUNDED`.

```
String relationID;
…
String myObjectsType = service.getType(myObject);
int rMin = service.getRelationMin
    (myObjectsType, relationID);
int rMax = service.getRelationMax
    (myObjectsType, relationID);
Vector values = myObject.get(relationID);
if(values.size() < rMin) {
    System.out.println("too few values");
} else if(rMax != Schema.CARDINALITY_UNBOUNDED &&
          values.size() > rMax) {
    System.out.println("too many values");
}
```

5.5.4 Allowed types for links

For a relation whose target type is reference (i.e., a link), you can discover what types of objects you are permitted to link to using that relation. To find out if a particular type can be linked to, call `Schema`'s `isAllowedType` method.

```
String carTypeID = "auto:car";
String engineTypeID = "auto:engine";
String powerSource = "auto:powerSource";
String seatTypeID = "auto:seat";
…
Schema schema = new Schema(service);
if(schema.isAllowedType
      (carTypeID, powerSource, seatTypeID)) {
    System.out.println("A car may not have a " + seatTypeID
+ " as a power source");
}
```

To get a list of the ID's of all allowed types for a given relation (whose target type is reference) on a given type, use the following idiom:

```
String allowed[] =
    schema.getAllowedTypes(schema.getType(typeID));
```

5.6 Search

The NMDS API provides limited search capabilities[2]. They can be used to locate objects based on various criteria.

5.6.1 Searching for objects with a relation value

To search for an object of a specific type that has a relation with a target that matches a specific value, use `RepositoryService`'s `matchTarget` method. There are two variants, one for string targets and one for reference targets (other target types will be supported in later releases). They each return an array of references to objects that match.

```
String personTypeID = "person";
Reference joes[] = service.matchTarget
  (personTypeID, "name:first", "Joe");
```

5.6.2 Searching in a container

Two variants of `matchTarget` called `matchTargetInContainer` allow you to limit your search to the contents of a particular container.

5.6.3 Searching for objects updated since a certain time

To find all objects updated at or since a given time, call `RepositoryService`'s `getUpdatedSince` method, passing it a `java.util.Date`. It will return an array of references to all objects updated at or since the time represented by the `java.util.Date`.

```
java.util.Date then = new java.util.Date();
try {
    Thread.currentThread().sleep(3000);
    Reference since[] = service.getUpdatedSince(then);
    System.out.println(since.length +
      " objects updated in the last 3 seconds");
} catch(InterruptedException x) {
    // do nothing
}
```

Update times are represented at millisecond resolution and are determined by the clock of the machine hosting the repository. In order to measure update times against your client, your client machine's clock should be synchronized to a reliable, external clock source. Updating an object takes time, and NMDS guarantees only that the update time recorded in the repository is between the time the update was initiated and the time it completed.

To find all versions of a particular object since a certain time, call a variant of `getUpdatedSince`, passing it a reference to the object in question.

```
Reference myObject;
…
```

[2] More extensive search capabilities are planned for later releases.

```
java.util.Date then = new java.util.Date();
try {
    Thread.currentThread().sleep(3000);
    Reference since[] = service.getUpdatedSince
      (then, myObject);
    if(since.length > 0) {
        System.out.println("my object was just updated");
    }
} catch(InterruptedException x) {
    // do nothing
}
```

5.7 Namespaces

Object and type ID's in the repository must be unique. Because this is a difficult requirement to meet when the repository has many distributed users, NMDS provides a namespace facility. Each ID can have a local part and a namespace part. The combination of both must be unique, but two ID's with a different namespace part can have the same local part and still be distinct.

To use namespaces, use the `org.nees.md.Namespace` utility class. To create an ID with a local part and a namespace part, call `Namespace`'s `id` method, passing it the namespace part and the local part:

```
String carFender = Namespace.id("auto", "fender");
…
String fenderGuitar = Namespace.id("guitar", "fender");
```

Alternatively, you can use Namespace to create many ID's with the same namespace part:

```
Namespace phone = new Namespace("phone");
String receiverID = phone.id("receiver");
String numberID = phone.id("number");
… etc …
```

To get the different parts of an ID, call `Namespace`'s `getNamespacePart` and `getLocalPart` methods. Continuing our previous example:

```
Namespace.getNamespacePart(numberID)
```

will return "`phone`", and

```
Namespace.getLocalPart(numberID)
```

will return "`number`".

6 Appendix A: example application

This example application connects to an NMDS service and prints a hierarchical listing of all objects contained in the root container or any of its children. For each object, it prints the type of the object, the object's title, and the time the object was last updated.

The application displays only the latest version of each object. To accomplish this, it calls `getLatestVersion` on each reference it retrieves from the service.

This application does not modify any of the objects in the repository.

```
import java.net.URL;

import org.nees.md.*;
import org.nees.md.service.*;
import org.nees.repo.*;
import org.nees.repo.service.*;
import org.nees.repo.axis.client.*;

public class ListContainers {
    RepositoryService repo;
    Schema schema;

    public ListContainers(RepositoryService r) {
        repo = r;
        schema = new Schema(r);
    }

    /**
     * Print out a listing of the containers and objects
     * in this container
     * @param c the container to start from
     */
    public void listContainers(Reference c)
      throws MetadataServiceException {
        listContainers(c, 0);
    }

    void indent(int n) {
        for(int i = 0; i < n; i++) {
            System.out.print("  ");
        }
    }

    void listContainers(Reference c, int level)
      throws MetadataServiceException {
        // get the latest version of the container
        c = repo.getLatestVersion(c);
        // get references to all the contained objects
        Reference contents[] = repo.getContents(c);
        // iterate over the contents
        for(int i = 0; i < contents.length; i++) {
            // get the latest version of each sub-object
            Reference o =
              repo.getLatestVersion(contents[i]);
```

```
            indent(level);
            // print out the type ID and title of the
object
            System.out.print(repo.getType(o) + " \""
              + repo.getTitle(o) + "\"");
            // print out last updated time
            System.out.print(" " + repo.getTimeUpdated(o));
            System.out.println();
            // if the object is itself a container, recur
            if(repo.isContainer(o)) {
                listContainers(o, level + 1);
            }
        }
    }

    public static void main(String args[])
      throws Exception {
        // the URL of the NMDS service
        String nmdsURL = args[0];

        // register a metadatafactory
        // (this is so we can call newReference below)
        MetadataFactory.setFactory
          (new MetadataFactoryImpl());

        // get an NMDS client
        NMDSClientFactory factory =
          new NMDSClientFactoryImpl
            (new URL(nmdsURL));
        RepositoryService repo =
          factory.newRepositoryService();

        ListContainers lc = new ListContainers(repo);

        // get the root container
        Reference root =
          MetadataFactory.getFactory().newReference
            ("nees:rootContainer");
        root = repo.getLatestVersion(root);

        // now list the contents of the root container
        lc.listContainers(root);
    }
}
```