UC Libraries Preservation Repository

System Design

Version 1.8.8
February 17, 2004

California Digital Library
University of California

The preservation repository system design detailed in this document represents the CDL's decisions in the *planning* phase of the project. The CDL is currently in the *development* phase of the project. During this phase changes might be made to the system design. These changes will not be represented in this document.

February 17, 2004

© UC Regents, 2003

1 OVERVIEW **5**

2 DESIGN APPROACH **6**

3 LANGUAGE AND PROTOCOLS **7**

4 THIRD-PARTY COMPONENTS **8**

5 DESIGN **9**

5.1 INFRASTRUCTURE 9
5.1.1 Component Registries *9*
5.1.2 Properties Manager *9*
5.1.3 Event Logger *9*
5.2 FUNCTIONAL SPECIFICATION 10
5.2.1 Overview *10*
5.2.2 Ingest Client *11*
5.2.3 Ingest *11*
5.2.3.1 Design Approach 13
5.2.3.2 Class IngestReceiver 13
5.2.3.2.1 Method init 13
5.2.3.2.2 Method doPost 13
5.2.3.3 Class IngestController 14
5.2.3.3.1 Method createObject 14
5.2.3.3.2 Method addVersion 14
5.2.3.3.3 Method replaceVersion 14
5.2.3.3.4 Method removeVersion 15
5.2.3.3.5 Method validateObject 15
5.2.3.4 Interface SIPParser 15
5.2.3.4.1 Method parse 15
5.2.3.5 Class SIPParser_METS 16
5.2.3.5.1 Method parse 16
5.2.3.6 Interface AIPValidator 16
5.2.3.6.1 Method validate 16
5.2.3.7 Class AIPValidator_BASIC 16
5.2.3.7.1 Method validate 16
5.2.3.8 Interface AIPIngestor 17
5.2.3.8.1 Summary of Methods 17
5.2.3.9 Class AIPIngestor_PRESERVE 17
5.2.3.10 Class IngestUtility 17
5.2.3.10.1 Method checkObject 17
5.2.3.10.2 Method getObjectStream 18
5.2.4 Access *19*
5.2.4.1 Class AccessReceiver 19
5.2.4.1.1 Method init 20
5.2.4.1.2 Method doPost 20
5.2.4.2 Class AccessController 20
5.2.4.2.1 Method getObject 20
5.2.4.3 Interface DIPFormatter 21
5.2.4.3.1 Method format 21
5.2.4.4 Class DIPFormatter_BASIC 21
5.2.4.4.1 Method format (Full) 21
5.2.4.4.2 Method format (Meta) 21
5.2.4.4.3 Method format (Summary) 22
5.2.5 Access Clients *22*
5.2.6 Security *22*
5.2.6.1 Interface SecurityManager 22

© UC Regents, 2003

5.2.6.1.1 Method checkAction 22
5.2.6.1.2 Method checkObjectRead 23
5.2.6.1.3 Method checkObjectWrite 23
5.2.6.2 SecurityManager_PRESERVE 23
5.2.7 Data Management 23
5.2.7.1 Interface RepoManager 23
5.2.7.1.1 Method createObject 23
5.2.7.1.2 Method addVersion 23
5.2.7.1.3 Method removeVersion 24
5.2.7.1.4 Method getObject 24
5.2.7.1.5 Method getCurrentVersionID 24
5.2.7.1.6 Method findObject 24
5.2.7.1.7 Method objectExists 25
5.2.7.2 Class RepoManager_MySQL 25
5.2.7.3 Interface AdminManager 25
5.2.7.3.1 Summary of Methods 25
5.2.7.4 Class AdminManager_MySQL 28
5.2.7.5 Interface IDManager 28
5.2.7.5.1 Method provideObjectID 28
5.2.7.6 Class IDManager_ARK 28
5.2.8 Archival Storage 28
5.2.8.1 Interface StorageManager 28
5.2.8.1.1 Method addObject 28
5.2.8.1.2 Method getObject 29
5.2.8.1.3 Method removeObject 29
5.2.8.2 Class StorageManager_SRB 29
5.2.9 Administration 30
5.2.9.1 Class AdminReceiver 30
5.2.9.1.1 Method init 31
5.2.9.1.2 Method doPost 31
5.2.9.2 Class AdminController 31
5.2.10 Administration Client 31
5.2.11 Infrastructure 32
5.2.11.1 Class Framework 32
5.2.11.1.1 Constructor 32
5.2.11.1.2 Method getSIPParser 33
5.2.11.1.3 Method getAIPValidator 33
5.2.11.1.4 Method getAIPIngestor 33
5.2.11.1.5 Method getDIPRetriever 33
5.2.11.1.6 Method getDIPFormatter 33
5.2.11.1.7 Method getSecurityManager 34
5.2.11.1.8 Method getRepoManager 34
5.2.11.1.9 Method getAdminManager 34
5.2.11.1.10 Method getIDManager 34
5.2.11.1.11 Method getStorageManager 34
5.2.11.1.12 Method getLogger 34
5.2.11.1.13 Method getProperty 34
5.2.11.1.14 Method getProperty 35
5.2.11.2 Interface Logger 35
5.2.11.2.1 Method logMessage 35
5.2.11.2.2 Method logError 35
5.2.11.3 Class FileLogger 35
5.2.11.4 Class Monitor 36
5.2.11.4.1 Constructor 36
5.2.11.4.2 Method start 36
5.2.11.4.3 Method stop 36
5.3 DATA MODEL 36
5.3.1 Submission Information Package (SIP) 36
5.3.1.1 Class SIPackage 37
5.3.2 Archival Information Package (AIP) 37
5.3.2.1 Class AIPackage 37

© UC Regents, 2003

5.3.3 Dissemination Information Package (DIP) ... 38
5.3.3.1 Class DIPackage ... 38
5.3.3.2 Class DIPackageFull ... 38
5.3.3.2.1 Constructors ... 38
5.3.3.3 Class DIPackageMeta ... 38
5.3.3.3.1 Constructors ... 38
5.3.3.4 Class DIPackageSummary ... 38
5.3.3.4.1 Constructors ... 39
5.3.4 Stored Data Model ... 40
5.3.4.1 Digital Object Repository Overview ... 40
5.3.4.2 Metadata Database – MySQL Implementation ... 41
5.3.4.2.1 Table DO_DIGITAL_OBJECT ... 41
5.3.4.2.2 Table DO_VERSION ... 41
5.3.4.2.3 Table DO_AGENT ... 42
5.3.4.2.4 Table DO_FILEGROUP ... 42
5.3.4.2.5 Table DO_FILE ... 42
5.3.4.2.6 Table DO_STRUCTMAP ... 42
5.3.4.2.7 Table DO_DESCRIPT_MD ... 43
5.3.4.2.8 Table DO_RIGHTS_MD ... 43
5.3.4.2.9 Table DO_TECH_MD ... 43
5.3.4.2.10 Table DO_SOURCE_MD ... 43
5.3.4.2.11 Table DO_DIGIPROV_MD ... 43
5.3.4.2.12 Table DO_BEHAVIOR_MD ... 43
5.3.4.3 Administration Database ... 44
5.3.4.4 Administration Database – MySQL Implementation ... 45
5.3.4.4.1 Table AD_ACCESS_GROUP ... 45
5.3.4.4.2 Table AD_PRODUCER ... 45
5.3.4.4.3 Table AD_CONSUMER ... 45
5.3.4.4.4 Table AD_USER ... 45
5.3.4.4.5 Table AD_ACCESS_GRP_MEMBER ... 46
5.3.4.4.6 Table AD_SUBMISSION ... 46
5.3.4.4.7 Table AD_COLLECTION ... 46
5.3.5 Control Information ... 46
5.3.5.1 Class ObjectID ... 46
5.3.5.1.1 Constructor ... 46
5.3.5.2 Class Request ... 47
5.3.5.2.1 Constructor ... 47
5.3.5.2.2 Method isAllowed ... 47
5.3.5.3 Class Query ... 47
5.3.5.3.1 Constructor ... 47
5.3.5.4 Class IngestResult ... 47
5.3.5.4.1 Constructor ... 48
5.3.5.5 Class AccessResult ... 48
5.3.5.5.1 Constructor ... 48
5.3.5.6 Class SIPParseResult ... 48
5.3.5.6.1 Constructor ... 48
5.3.5.7 Class SecurityResult ... 48
5.3.5.7.1 Constructor ... 48
5.3.5.8 Class RepoResult ... 49
5.3.5.8.1 Constructor ... 49
5.3.5.9 Class AdminResult ... 49
5.3.5.9.1 Constructor ... 49
5.3.5.10 Class StorageResult ... 49
5.3.5.10.1 Constructor ... 49
5.4 INTERFACES ... 50
5.4.1 Ingest API ... 50
5.4.1.1 Method createObject ... 50
5.4.1.2 Method addVersion ... 50
5.4.1.3 Method replaceVersion ... 51
5.4.1.4 Method removeVersion ... 52
5.4.1.5 Method validateObject ... 52

© UC Regents, 2003

5.4.2 Access API 52
5.4.2.1 Method getObject 52
5.4.3 Administration API 53
5.4.3.1 Repository Administration API.. 53
5.4.3.1.1 Method getObjectMetadata.. 53
5.4.3.1.2 Method getObjectSummary.. 53
5.4.3.2 General Administration API 55
5.4.3.2.1 Method createProducer.. 55
5.4.3.2.2 Method updateProducer.. 55
5.4.3.2.3 Method removeProducer 55
5.4.3.2.4 Method getProducer... 55
5.4.3.2.5 Method getProducerList 55
5.4.3.2.6 Method createConsumer... 55
5.4.3.2.7 Method updateConsumer.. 55
5.4.3.2.8 Method removeConsumer... 55
5.4.3.2.9 Method getConsumer.. 55
5.4.3.2.10 Method getConsumerList 55
5.4.3.2.11 Method createAccessGroup.. 55
5.4.3.2.12 Method updateAccessGroup... 55
5.4.3.2.13 Method removeAccessGroup 55
5.4.3.2.14 Method getAccessGroup 55
5.4.3.2.15 Method getAccessGroupList 55
5.4.3.2.16 Method addAccessGroupMember 55
5.4.3.2.17 Method removeAccessGroupMember 55
5.4.3.2.18 Method getAccessGroupMemberList... 55
5.4.3.2.19 Method getSubmission 55
5.4.3.2.20 Method createSubmission... 55
5.4.3.2.21 Method updateSubmission.. 55
5.4.3.2.22 Method removeSubmission 55
5.4.3.2.23 Method getSubmissionList 55
5.4.3.2.24 Method getSubmission 55
5.4.3.2.25 Method getCollection 55
5.4.3.2.26 Method createCollection... 55
5.4.3.2.27 Method updateCollection.. 55
5.4.3.2.28 Method removeCollection 55
5.4.3.2.29 Method getCollectionList 55
5.4.3.2.30 Method getCollection 56

© UC Regents, 2003

1 Overview

The California Digital Library (CDL), on behalf of the University of California Libraries, has undertaken the task of building a repository for digital objects that will:

- Preserve digital objects and their metadata for long periods of time
- Control access to preserved objects
- Assign reliable, persistent identifiers to preserved objects
- Administer objects and produce management reports

This document describes the *Preservation Repository* (*PR*), a system that will enable the UC Libraries to provide these services. This is a working document that will be revised as detailed design decisions are made and components are implemented or acquired.

© UC Regents, 2003

2 Design Approach

The primary design goal of the design for the Preservation Repository is to meet the needs of the UC Libraries for a reliable and secure repository. An ancillary goal is to create a system that is flexible and extensible, and which contributes to future related applications at CDL.

CDL is in the process of defining a *Common Framework* (CF) for services and supporting systems. To the extent possible – given the overlapping timelines of the CF design and the Preservation Repository design - the design of PR will be consistent with the concepts of the CF.

PR will contribute core infrastructure components to the CF that will be useful beyond the specific needs of the Preservation Repository, and that will make the PR itself easily extended and enhanced in the future. In a sense, the PR will be the foundry for pieces of the CF that it needs, and care will be taken to make these components adaptable for use with future CDL systems with similar needs.

The Common Framework will be based on the concept of *services.* A service will use the CF to deliver a specific set of functions to designated communities of content producers and consumers. Each service will be implemented as a series of modules that will plug into the framework. The Preservation Repository will be the first such service. Development will therefore produce two distinct sets of software:

- Core Common Framework components needed by the PR
- Service components that implement the specific functionality of the PR

The system will be open and extensible. Automated tasks will be broken into well-defined functional components presenting clearly defined interfaces. The architecture will follow the *Reference Model for an Open Archival Information System (OAIS)* published by the Consultative Committee for Space Data Systems (CCSDS), which is also the foundation for the Common Framework.

Individual components will be implemented through:

- The use of existing in-house components, with in-house developed wrappers
- In-house development, leveraging existing in-house models or prototypes to the extent possible
- The use of third party components, with in-house developed 'wrappers' to integrate them into the PR/CF framework

Components will be loosely coupled to enable distribution of the system across servers and to facilitate load balancing.

The system will be developed using languages and third-party components that are – to the extent possible – platform-independent.

A database management system (DBMS) will be used for the storage and management of metadata. DBMS security will be masked through a data management service to allow for the use of different DBMS products.

© UC Regents, 2003

3 Language and Protocols

Components that will be developed in-house will be implemented in Java J2EE. The following Java toolkits will be used:

- Java Web Services Developer Pack

The following protocols will be used:

- ARK
- METS
- XML
- SOAP

© UC Regents, 2003

4 Third-Party Components

The following third-party components may be utilized by the Preservation Repository:

- *Storage Resource Broker (SRB),* developed by the San Diego Supercomputer Center (SDSC) at the University of California at San Diego, may be used for archive storage (see section 5.2.4) and for portions of data management (see section 5.2.7)
- *JARGON*, a Java client API for SRB developed by SDSC, may be used in conjunction with SRB.
- *Shibboleth*, a project of Internet2/MACE, may be used to implement security control (see section 5.2.4.4.2)
- *MySQL*, a relational database management system developed by MySQL AB, may be used for data management . It may be used in conjunction with SRB for management of storage-related metadata and separately for management of other types of data.

© UC Regents, 2003

5 Design

5.1 Infrastructure

An application infrastructure will be developed to provide basic services to the functional components of the system. The services provided will include:

- C*omponent registries* that will enable new functions or protocol handlers to be added dynamically to the system
- A *properties manager* that will centralize and simplify the configuration of functional components
- An *event logger* that will centralize the recording of errors and informational messages

5.1.1 Component Registries

Component registries will provide a means for identifying components to the system and for the system to automatically locate components and execute their methods in the appropriate context. For example, a registry of Submission Information Package (SIP) parsers and an associated dispatcher will be created for the Ingest Function that will allow an appropriate parser to be executed based upon the format of a SIP stream (e.g., METS.)

Component registries will be implemented in the Common Framework class *org.cdlib.framework.utility.Framework* (see section 5.2.11.1.)

5.1.2 Properties Manager

A central properties manager will provide a means for specifying configuration information in a single place and for obtaining this information easily and in a consistent manner.

The properties manager will be implemented in the CF class *org.cdlib.framework.utility.Framework* (see section 5.2.11.1.)

5.1.3 Event Logger

An event logger component will enable functional components to send information or error messages to a centrally maintained suite of event logs.

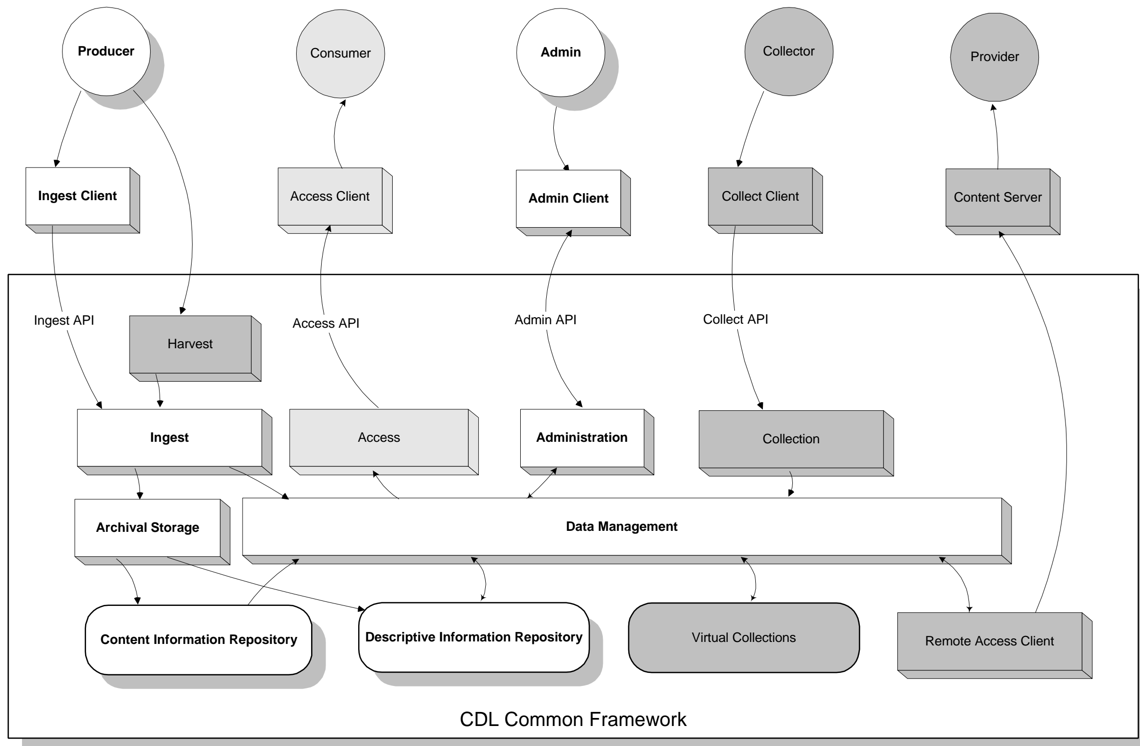
The event logger will be implemented as a CF class in package *org.cdlib.framework.utility*.

© UC Regents, 2003

5.2 Functional Specification

5.2.1 Overview

The following diagram illustrates the high-level functional components of the Preservation Repository and their relationship to the Common Framework.



White boxes indicate the major functional components of the PR. Those appearing inside of the "CDL Common Framework" are components that will be implemented with "common purpose" in mind. Those outside of the CF boundary will be PR-only, but will conform to common interface specifications for clients of the CF.

White circles indicate the categories of user communities for PR.

White boxes with rounded corners represent data stores that will be developed for PR.

Light grey objects (i.e. those relating to Access) indicate components that are relevant to PR, but only minimally. These will be implemented as common purpose, but will only address some parts of the Common Framework.

Dark grey objects indicate portions of the CF that are not relevant to PR. They are included to illustrate the relationship of PR to the CF as a whole. **These components will not be implemented in the Preservation Repository**.

The CF will also contain a pervasive Security component, which is not illustrated in the diagram. PR will include this component.

The following sections describe the functional components in more detail.

© UC Regents, 2003

5.2.2 Ingest Client

The Ingest Client will interact directly with a producer of digital objects, prepare objects for submission to the preservation repository and initiate the submission of objects by communicating with the Ingest (server) component (see section 5.2.3). It will also interact with the Ingest component to:

- Establish a digitial object in the repository and obtain its identifier
- Request validation of an object prior to actual submission
- Add an initial or new version of a digital object
- Remove a version of a digital object
- Replace a version of an object

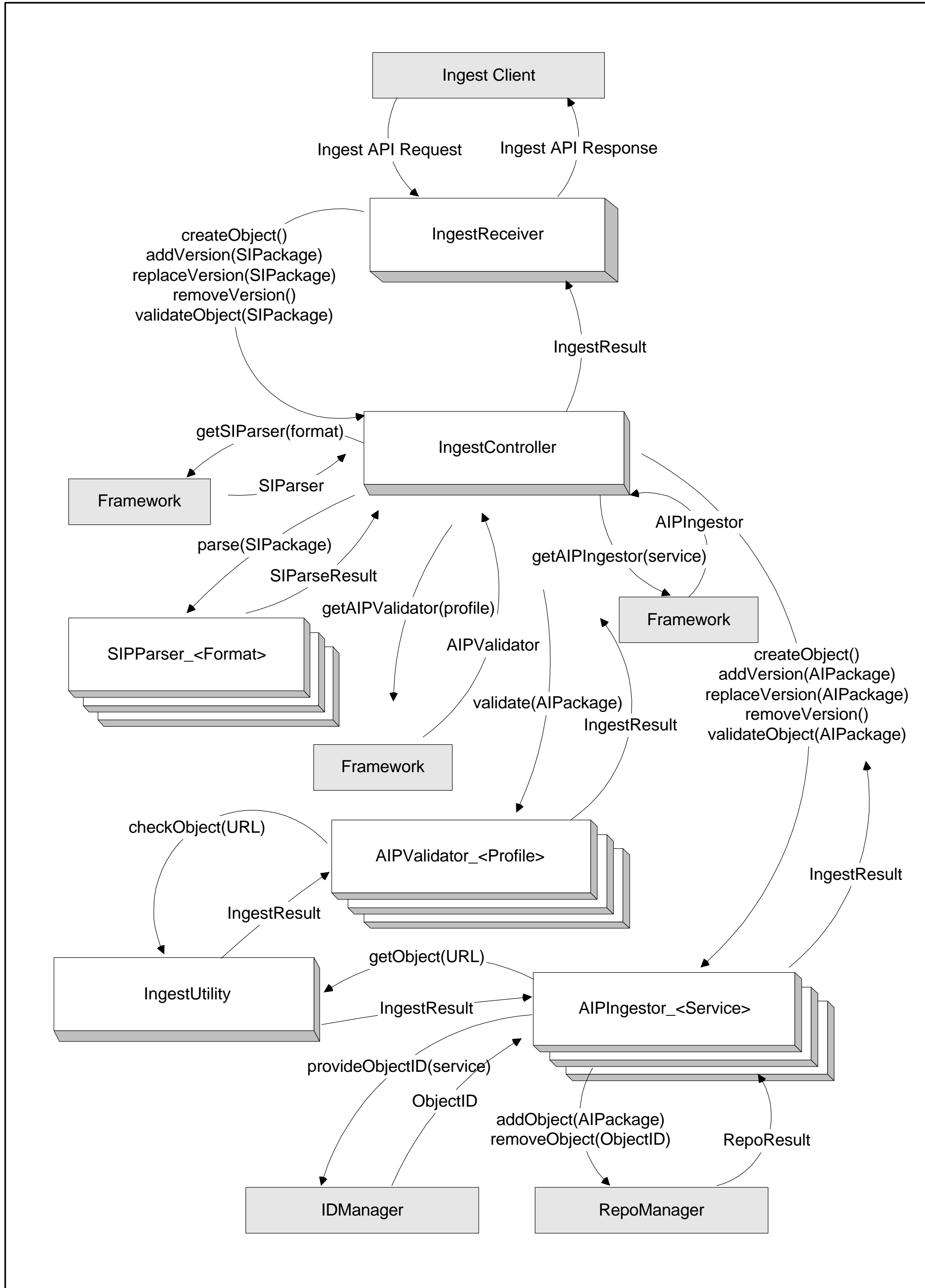
All communications between the Ingest Client and the Ingest component will be through the Ingest API (see section 5.4.1).

It is proposed that the current CDL ingest service, *voro*, be re-purposed as the Ingest Client in the Common Framework.

5.2.3 Ingest

The Ingest component will enable a producer of digital objects to store objects in the Preservation Repository and to manage digital objects previously stored. The function will receive Ingest API requests coming from the Ingest Client and translate the requests into actions for the Archival Storage and Data Management components of the system. The following diagram illustrates the sub-components of Ingest and the message flows among components and sub-components.

© UC Regents, 2003



Ingest will handle all methods defined in the Ingest API (see section 5.4.1.)

5.2.3.1 Design Approach

- HTTP transport for flexibility and support of diverse clients
- SOAP-based request/response message model for reliable interchange of control information
- Use of component registration for parsers and ingestors, for maximum adaptability

5.2.3.2 Class IngestReceiver

Class *IngestReceiver* will receive from an Ingest Client an HTTP-encapsulated stream of one or more SOAP-formatted Ingest API requests. It will instantiate objects to handle the requests, invoke handling methods on the objects and return response messages to the Ingest Client.

The class will be implemented as a Common Framework component in package *org.cdlib.framework.ingest*. It will be implemented as a Java servlet, extending *class javax.servlet.http.HTTPServlet*.

5.2.3.2.1 Method init

Signature:

```
protected void init(ServletConfig config)
```

This method will create a new *Framework* (see section 5.2.11.1) object and store it as a member variable. Servlet parameters will provide the name(s) of properties file(s) that will be passed to the *Framework* constructor.

5.2.3.2.2 Method doPost

Signature:

```
protected void doPost(HttpServletRequest req,
                      HttpServletResponse resp)
```

This method will:

1. Process a stream containing messages conforming to request messages in the Ingest API specification (see section 5.4.1) The stream containing the Ingest API messages will be expected to be present in the body portion of the HTTP POST request.
2. For each request message in the stream:
    1) Convert the message into a *SIPackage* object (see section 5.3.1.1)
    2) Construct a *Request* object (see section 5.3.5.2)
    3) Verify that the requested action is allowed for the producer, using the *SecurityManager* for the requested service (see section 5.3.5.2)
    4) Invoke the appropriate method of the *IngestController* class (see section 5.2.3.3) to perform the action
    5) Receive an *IngestResult* containing status and other information from the *IngestController* and construct and queue a response message to the client that communicates the results

© UC Regents, 2003

5.2.3.3 Class IngestController

Class *IngestController* will control the sequence of events necessary for processing a request. A separate method will exist for each type of request defined in the Ingest API specification (see section 5.4.1.) All methods will return an *IngestResult* (see section 5.2.7.3) object.

The class will be implemented as a Common Framework component in package *org.cdlib.framework.ingest*.

5.2.3.3.1 Method createObject

Signature:

```
public IngestResult createObject(
        Framework fw, Request request)
```

This method will generate an object ID and create an 'empty' digital object (i.e. and object with no versions), as follows:

1. Find an *AIPIngestor* corresponding to the *service* specified in *request* by invoking the *getAIPIngestor* method of *fw*
2. Obtain an object ID and create an empty digital object by invoking the *create* method of the *AIPIngestor*

5.2.3.3.2 Method addVersion

Signature:

```
public IngestResult addVersion(
        Framework fw, Request request, SIPackage sip)
```

This method will add a new version of a digital object, as follows:

1. Find a *SIPParser* to handle the *format* specified in the *SIPackage* by invoking the *getSIPParser* method of *fw*
2. Convert the *SIPackage* into an *AIPackage* by invoking the *parse* method of the *SIPParser*
3. Find an *AIPValidator* corresponding to the *profile* specified in the *AIPackage* by invoking the *getAIPValidator* method of *fw*
4. Validate the *AIPackage* by invoking the validate method of the *AIPValidator*
5. Find an *AIPIngestor* corresponding to the *service* specified in *request* by invoking the *getAIPIngestor* method of *fw*
6. Store the validated *AIPackage* by invoking the *add* method of the *AIPIngestor*.

5.2.3.3.3 Method replaceVersion

```
public IngestResult replaceVersion(
        Framework fw, Request request, SIPackage sip)
```

This method will replace the current version of a digital object or, if there is no current version, add a new version. It will operate as follows:

1. Find a *SIPParser* to handle the *format* specified in the *SIPackage* by invoking the *getSIPParser* method of *fw*

© UC Regents, 2003

2. Convert the *SIPackage* into an *AIPackage* by invoking the *parse* method of the *SIPParser*
3. Find an *AIPValidator* corresponding to the *profile* specified in the *AIPackage* by invoking the *getAIPValidator* method of *fw*
4. Validate the *AIPackage* by invoking the validate method of the *AIPValidator*
5. Find an *AIPIngestor* corresponding to the *service* specified in *request* by invoking the *getAIPIngestor* method of *fw*
6. Remove the current version of the object and add the resulting *AIPackage* by invoking the *replace* method of the *AIPIngestor*

5.2.3.3.4 Method removeVersion

Signature:

```
public IngestResult removeVersion(
        Framework fw, Request request, ObjectID objectID )
```

This method will:

1. Find an *AIPIngestor* corresponding to the *service* specified in *request* by invoking the *getAIPIngestor* method of *fw*
2. Remove the current version of the digital object identified by the objectID parameter by invoking the *remove* method of the *AIPIngestor*

5.2.3.3.5 Method validateObject

Signature:

```
public IngestResult validateObject(
        Framework fw, Request request, SIPackage sip)
```

This method will:

1. Find a *SIPParser* to handle the *format* specified in the *SIPackage* by invoking the *getSIPParser* method of *fw*
2. Convert the *SIPackage* into an *AIPackage* by invoking the *parse* method of the *SIPParser*
3. Find an *AIPValidator* corresponding to the *profile* specified in the *AIPackage* by invoking the *getAIPValidator* method of *fw*
4. Validate the *AIPackage* by invoking the validate method of the *AIPValidator*

5.2.3.4 Interface SIPParser

This interface will define standard SIP parser behavior. It will be defined as a Common Framework component in package *org.cdlib.framework.ingest*.

The appropriate implementation class for a *SIPackage* will be obtained via the *getSIPParser* method of the Framework class (see section 5.2.11.1.2.) The *format* property of the *SIPackage* will be used by the Framework to determine the appropriate *SIPParser*.

5.2.3.4.1 Method parse

Signature:

© UC Regents, 2003

```
static public SIPParseResult parse(
        Framework fw, SIPackage sip)
```

Implementers of this method will convert a *SIPackage* into an *AIPackage*.

5.2.3.5 Class SIPParser_METS

This class will implement interface *SIPParser* and will be a registered *SIPParser*. It will be used to parse *SIPackages* in METS format.

5.2.3.5.1 Method parse

This implementation of the *parse* method will use the *Java XSLT API* (part of *Java Web Services Developer Pack*) and an XSLT style sheet to copy metadata that is already in METS format from a *SIPackage* to an *AIPackage*. The stylesheet location and name will be specified in the properties file (see section 5.2.11.1) as follows:

Property	*Description*	*Example*
styleSheet.path	Path (relative to resource path root) to directory containing xslt style sheets	**/stylesheets**
styleSheet.METS.parse	Name of style sheet for parsing METS	**mets2mets.xslt**

5.2.3.6 Interface AIPValidator

This interface will define standard AIP validator behavior. It will be defined as a Common Framework component in package *org.cdlib.framework.ingest*.

The appropriate implementation class for an *AIPackage* will be obtained via the *getAIPValidator* method of the Framework class (see section 5.2.11.1.3.) The *profile* property of the *AIPackage* will be used by the Framework to determine the appropriate *AIPValidator*.

5.2.3.6.1 Method validate

Signature:

```
static public IngestResult validate(
        Framework fw, AIPackage aip)
```

Implementers of this method will test that an *AIPackage* conforms to an agreed-upon specification for submission.

5.2.3.7 Class AIPValidator_BASIC

This class will implement the *AIPValidator* interface and will be a registered AIPValidator. It will be implemented as a Preservation Repository component in package *org.cdlib.preservation.ingest*.

5.2.3.7.1 Method validate

Signature:

```
static public IngestResult validate(
        Framework fw, AIPackage sip)
```

This method will verify following about the *AIPackage*:

- There is at least one *fileGrp* element
- There is at least one *file* element in each fileGrp

© UC Regents, 2003

- There is at least one *FLocat* (file location) element in each *file*
- Each *FLocat* has an *xlink:href* and a *LocType* attribute
- The object referred to by each *xlink:href* attribute is accessible (using the *checkObject* method of *ImageUtility*)

5.2.3.8 Interface AIPIngestor

This interface will define standard AIP ingestor behavior. It will be defined as a Common Framework component in package *org.cdlib.framework.ingest*.

The appropriate implementation class will be obtained via the *getAIPIngestor* method of the *Framework* class (see section 5.2.11.1.3) The *service* property of the *AIPackage* will be used by the Framework to determine the appropriate *AIPIngestor* implementation.

5.2.3.8.1 Summary of Methods

Signature:

```
static public IngestResult createObject(
        Framework fw, Request request)

static public IngestResult addVersion(
        Framework fw, Request request, AIPackage aip)

static public IngestResult replaceVersion(
        Framework fw, Request request, AIPackage aip)


static public IngestResult removeVersion(
        Framework fw, Request request, AIPackage aip)


static public IngestResult validateObject(
        Framework fw, Request request, AIPackage aip)
```

AIP ingestors for specific services (e.g., PRESERVE) will provide implementations of these methods that apply the services' business rules.

5.2.3.9 Class AIPIngestor_PRESERVE

This class will implement the *AIPIngestor* interface for the Preservation Repository. It will reside in package *org.cdl.preservation.ingest* and will be registered as an AIP ingestor in the *Framework*.

5.2.3.10 Class IngestUtility

This class will contain methods of general use to the Ingest function. It will reside in package *org.cdl.preservation.ingest.*

5.2.3.10.1 Method checkObject

Signatures:

```
static public IngestResult checkObject(
        Framework fw, URL url)

static public IngestResult checkObject(
        Framework fw, String fileName)
```

© UC Regents, 2003

These methods will check for the existence of an object at a specified URL or having a specified file name in the local file system.

5.2.3.10.2 Method getObjectStream

Signatures:

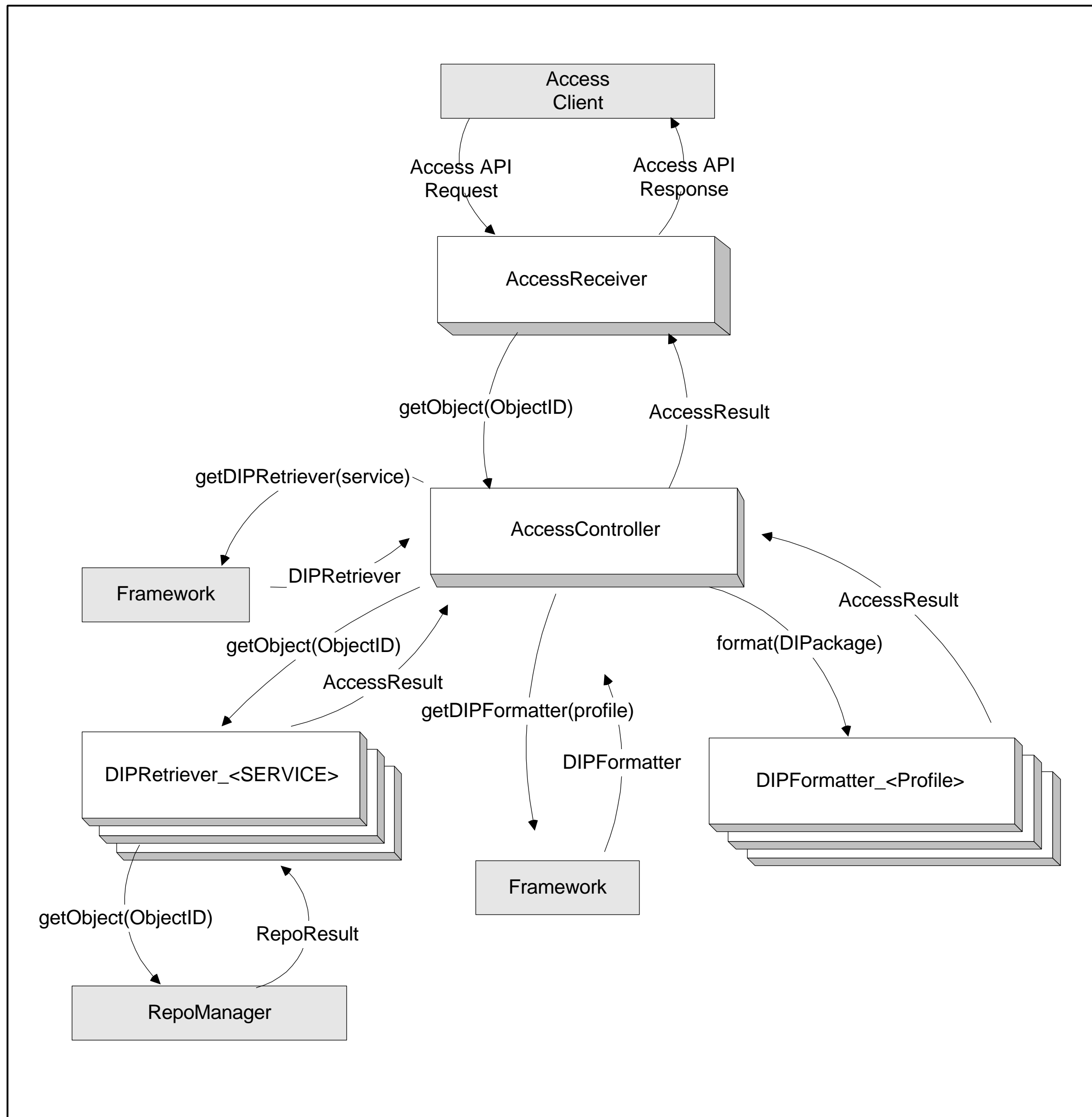
```
static public InputStream getObjectStream(
        Framework fw, URL url)

static public InputStream getObjectStream(
        Framework fw, String fileName)
```

These methods will return the contents of a specified URL or a file in the local file system as a byte input stream.

© UC Regents, 2003

5.2.4 Access

The Access component will enable a consumer to retrieve digital objects from the Preservation Repository. The function will receive Access API requests coming from an Access Client and translate the requests into actions for the Data Management component of the system. The following diagram illustrates the sub-components of Access and the message flows among components and sub-components.



5.2.4.1 Class AccessReceiver

Class *AccessReceiver* will receive from an Access Client an HTTP-encapsulated stream of one or more SOAP-formatted Access API requests. It will instantiate objects to handle the requests, invoke handling methods on the objects and return response messages to the Access Client.

The class will be implemented as a Common Framework component in package *org.cdlib.framework.access*. It will be implemented as a Java servlet, extending *class javax.servlet.http.HTTPServlet*.

5.2.4.1.1 Method init

Signature:

```
protected void init(ServletConfig config)
```

This method will create a new *Framework* (see section 5.2.11.1) object and store it as a member variable. Servlet parameters will provide the name(s) of properties file(s) that will be passed to the *Framework* constructor.

5.2.4.1.2 Method doPost

Signature:

```
protected void doPost(HttpServletRequest req,
                      HttpServletResponse resp)
```

This method will:

1. Process a stream containing messages conforming to request messages in the Access API specification (see section 5.4.1) The stream containing the Access API messages will be expected to be present in the body portion of the HTTP POST request.

2. For each request message in the stream:

    1) Convert the message into a *Query* object (see section 5.3.1.1)

    2) Construct a *Request* object (see section 5.3.5.2)

    3) Verify that the requested action is allowed for the consumer, using the *SecurityManager* for the requested service (see section 5.3.5.2)

    4) Invoke the appropriate method of the *AccessController* class (see section 5.2.4.2) to perform the action

    5) Receive an *AccessResult* containing status and other information from the *AccessController*

5.2.4.2 Class AccessController

Class AccessController will control the sequence of events necessary for processing an access request. A separate method will exist for each type of request defined in the Access API specification (see section 5.4.2.) All methods will return an *AccessResult* (see section 5.3.5.5) object.

The class will be implemented as a Common Framework component in package *org.cdlib.framework.access*.

5.2.4.2.1 Method getObject

Signature:

```
public AccessResult getObject(
        Framework fw, Request request, Query query)
```

This method will:

1. Find a *DIPRetriever* corresponding to the *service* specified in the *Query* by invoking the *getDIPRetriever* method of *fw*

2. Retrieve a *DIPackage* for the requested object by invoking the *getObject* method of the *DIPRetriever*

3. Store the retrieved *DIPackage* in an *AccessResult* object and return it

5.2.4.3 Interface DIPFormatter

This interface will define standard behavior for classes that transform a DIPackage into a specific output format based on a profile. It will be defined as a Common Framework component in package *org.cdlib.framework.access*.

The appropriate implementation class for will be obtained via the *getDIPFormatter* method of the Framework class (see section). The *profile* property of the *DIPackage* will be used by the Framework to determine the appropriate *DIPFormatter*.

5.2.4.3.1 Method format

Signature:

```
static public AccessResult format(
        Framework fw, DIPackageFull dip)

static public AccessResult format(
        Framework fw, DIPackageMeta dip)

static public AccessResult format(
        Framework fw, DIPackageSummary dip)
```

Implementers of this method will transform the contents of a full, metadata-only, or summary *dip* using XSLT and/or other tools.

5.2.4.4 Class DIPFormatter_BASIC

This class will implement the *DIPFormatter* interface and will be a registered *DIPFormatter*. It will be implemented as a Common Framework component in package *org.cdlib.framework.access*.

5.2.4.4.1 Method format (Full)

This implementation of the *format* method will use the *Java XSLT API* (part of *Java Web Services Developer Pack*) and an XSLT style sheet to format a *DIPackageFull*. The stylesheet location and name will be specified in the properties file (see section 5.2.11.1) as follows:

Property	*Description*	*Example*
styleSheet.path	Path (relative to resource path root) to directory containing xslt style sheets	**/stylesheets**
styleSheet.format.BASIC.full	Name of style sheet for formatting a BASIC *DIPPackageFull*	**formatBasicFull.xslt**

5.2.4.4.2 Method format (Meta)

This implementation of the *format* method will format a *DIPackageMeta*. The stylesheet location and name will be specified in the properties file (see section 5.2.11.1) as follows:

Property	*Description*	*Example*
styleSheet.path	Path (relative to resource path root) to directory containing xslt style sheets	**/stylesheets**
styleSheet.format.BASIC.meta	Name of style sheet for formatting a BASIC *DIPPackageMeta*	**formatBasicMeta.xslt**

© UC Regents, 2003

5.2.4.4.3 Method format (Summary)

This implementation of the *format* method will format a *DIPackageSummary*. The stylesheet location and name will be specified in the properties file (see section 5.2.11.1) as follows:

Property	*Description*	*Example*
`styleSheet.path`	Path (relative to resource path root) to directory containing xslt style sheets	`/stylesheets`
`styleSheet.format.BASIC.summary`	Name of style sheet for formatting a BASIC *DIPPackageSummary*	`formatBasicSummary.xslt`

5.2.5 Access Clients

Access clients will interact directly or indirectly with a consumer of digital objects and communicate with the Access function (see section 5.2.4) to:

- Retrieve complete digital objects (metadata and content)
- Retrieve metadata only for a digital object

All communication between access clients and the Access component will be through the Access API (see section 5.4.2).

Current CDL applications will be reviewed to determine if they can be modified to serve the needs of the Preservation Repository.

5.2.6 Security

The Security component will enable establishment and enforcement of rules that govern the ability of producers to create and delete digital objects, and the ability of consumers to obtain copies of stored objects or metadata associated with objects.

Note: For PR, the producer of an object will be the only allowable consumer of the object.

5.2.6.1 Interface SecurityManager

This interface will define standard behavior for a security manager. Each defined *service* will include an implementation of this interface that will enforce its set of security rules. It will reside in package *org.cdlib.framework.security*.

The appropriate implementation class will be obtained via the *getSecurityManager* method of the *Framework* class (see section 5.2.11.1.5) The *service* property of a request will be used by the Framework to identify the appropriate *SecurityManager* implementation.

5.2.6.1.1 Method checkAction

Signature:

```
public SecurityResult checkAction(
        Framework fw, Request request)
```

An implementation of this method will determine if a *Request* is permitted based on the action being requested and the identity of the requestor.

© UC Regents, 2003

5.2.6.1.2 Method checkObjectRead

Signature:

```
public SecurityResult checkObjectRead(
        Framework fw, Request request, ObjectID objectID)
```

An implementation of this method will determine if the requestor is allowed to read the contents of an object.

5.2.6.1.3 Method checkObjectWrite

Signature:

```
public SecurityResult checkObjectWrite(
        Framework fw, Request request, ObjectID objectID)
```

An implementation of this method will determine if the requestor is allowed to create or purge an object.

5.2.6.2 SecurityManager_PRESERVE

This class will implement the *SecurityManager* interface for the Preservation Respository. It will reside in package *org.cdlib.preservation.security*.

5.2.7 Data Management

The Data Management component will provide services to the Ingest, Administration and Access components for populating, maintaining, indexing and accessing objects and metadata in the Preservation Repository and for synchronizing metadata and stored objects in the repository. It will also provide the Administration component with functions for maintaining and accessing administrative data.

5.2.7.1 Interface RepoManager

Interface *RepoManager* will define the services that will be provided for managing objects and metadata. It will reside in package *org.cdlib.framework.dataManagement*.

5.2.7.1.1 Method createObject

Signature:

```
public RepoResult createObject(Framework fw, AIPackage aip)
```

This method will:

1. Obtain an Object ID (ARK Name) for the object
2. Store basic object information in the metadata database
3. Return the Object ID of the object in the *RepoResult*

5.2.7.1.2 Method addVersion

Signature:

```
public RepoResult addVersion(Framework fw, AIPackage aip)
```

This method will:

© UC Regents, 2003

1. Determine if an object exists with matching Object ID or altObjectID; if not, obtain an Object ID (ARK Name) for the object and create the object
2. Store the altObjectID if one is provided
3. Store the object(s) in the AIPackage's objecURIList, via the Archival Storage component
4. Create a new version, incrementing the object's current version
5. Store metadata XML fragments
6. Store the structured metadata, with references to object(s) and metadata XML fragment(s), in the repository database
7. Return the Object ID of the object in the *RepoResult*

5.2.7.1.3 Method removeVersion

Signature:

```
public RepoResult removeVersion(Framework fw, AIPackage aip)
```

This method will:

1. Determine the current version of the digital object
2. Get the storage ID's of all component objects for the version
3. Remove the current version's metadata
4. Remove the current version's object components, via the Archival Storage function

5.2.7.1.4 Method getObject

Signature:

```
public RepoResult getObject(Framework fw, ObjectID objectID)
```

This method will retrieve a specified object and its metadata. It will:

1. Retrieve the metadata from the repository database
2. Retrieve object(s) via the Archival Storage component
3. Construct a *DIPackage* (see section 5.3.3.1) and return it in the *RepoResult*

5.2.7.1.5 Method getCurrentVersionID

Signature:

```
public int getCurrentVersionID(Framework fw, ObjectID objectID)
```

This method will return the current version ID (ie. the sequence) of a specified object.

5.2.7.1.6 Method findObject

Signature:

```
public ObjectID findObject(
        Framework fw,
        String producerID,
        String altObjectID)
```

© UC Regents, 2003

This method will find an object based on its *producerID* and its *altObjectID*.

5.2.7.1.7 Method objectExists

Signature:

```
public boolean objectExists(Framework fw, ObjectID objectID)
```

This method will report whether or not an object exists with a specified ID.

5.2.7.2 Class RepoManager_MySQL

This class will implement the *RepoManager* interface using MySQL.

5.2.7.3 Interface AdminManager

Interface *AdminManager* will define the services that will be provided for managing administrative data. It will reside in package *org.cdlib.framework.dataManagement*.

5.2.7.3.1 Summary of Methods

Signatures:

```
public AdminResult createProducer(
        Framework fw,
        String producerID,
        String name,
        String address1,
        String address2,
        String city,
        String state,
        String postalCode,
        String country,
        String contactName,
        String contactTitle,
        String contactEmail,
        String contactPhone,
        String authCode)

public AdminResult updateProducer(
        Framework fw,
        String producerID,
        String name,
        String address1,
        String address2,
        String city,
        String state,
        String postalCode,
        String country,
        String contactName,
        String contactTitle,
        String contactEmail,
        String contactPhone,
        String authCode)

public AdminResult removeProducer(
        Framework fw,
        String producerID)
```

© UC Regents, 2003

```
public AdminResult getProducer(
        Framework fw,
        String producerID)

public AdminResult getProducerList(
        Framework fw)

public AdminResult createConsumer(
        Framework fw,
        String consumerID,
        String name,
        String authCode)

public AdminResult updateConsumer(
        Framework fw,
        String consumerID,
        String name,
        String authCode)

public AdminResult removeConsumer(
        Framework fw,
        String consumerID)

public AdminResult getConsumer(
        Framework fw,
        String consumerID)

public AdminResult getConsumerList(
        Framework fw)

public AdminResult createAccessGroup(
        Framework fw,
        String accessGroupID,
        String name)

public AdminResult updateAccessGroup(
        Framework fw,
        String accessGrouID,
        String name)

public AdminResult removeAccessGroup(
        Framework fw,
        String accessGrouID)

public AdminResult getAccessGroup(
        Framework fw,
        String accessGrouID)

public AdminResult getAccessGroupList(
        Framework fw)


public AdminResult addAccessGroupMember(
        Framework fw,
        String accessGroupID,
        String memberID,
```

© UC Regents, 2003

```
        String memberType)

public AdminResult removeAccessGroupMember(
        Framework fw,
        String accessGroupID,
        String accessGroupMember)

public AdminResult createSubmission(
        Framework fw,
        String agreementID,
        String producerID,
        String description,
        String contactName,
        String contactTitle,
        String contactEmail,
        String contactPhone)

public AdminResult updateSubmission(
        Framework fw,
        String agreementID,
        String producerID,
        String description,
        String contactName,
        String contactTitle,
        String contactEmail,
        String contactPhone)

public AdminResult removeSubmission(
        Framework fw,
        String agreementID)

public AdminResult getSubmission(
        Framework fw,
        String agreementID)

public AdminResult getSubmissionList(
        Framework fw)

public AdminResult createCollection(
        Framework fw,
        String collectionID,
        String agreementID,
        String  name,
        String description)

public AdminResult updateCollection(
        Framework fw,
        String collectionID,
        String agreementID,
        String  name,
        String description)

public AdminResult removeCollection(
        Framework fw,
        String collectionID)

public AdminResult getCollection(
```

© UC Regents, 2003

```
        Framework fw,
        String collectionID)

public AdminResult getCollectionList(
        Framework fw)
```

5.2.7.4 Class AdminManager_MySQL

This class will implement the *AdminManager* interface using MySQL.

5.2.7.5 Interface IDManager

Interface *IDManager* will define the interface for generating unique identifiers for digital objects. It will reside in package *org.cdlib.framework.dataManagement*.

5.2.7.5.1 Method provideObjectID

This method will provide a unique object identifier.

Signature:

```
public ObjectID provideObjectID(
        String producerID, String serviceID)
```

5.2.7.6 Class IDManager_ARK

Class *IDManager_ARK* will implement interface IDManager by providing a wrapper for the current CDL ARK service. It will reside in package *org.cdlib.framework.dataManagement*.

Identifiers generated by this class will conform to the ARK Name specification.

Issue: Should the Preservation Repository be represented as an ARK Name Assigning Authority Number (NAAN)?

5.2.8 Archival Storage

The Archival Storage component will handle the storage, replication and retrieval of binary objects.

5.2.8.1 Interface StorageManager

Interface *StorageManager* will define the services that will be provided for managing, storing and retrieving binary objects. It will reside in package *org.cdlib.framework.archivalStorage*.

5.2.8.1.1 Method addObject

Signature:

```
public StorageResult addObject(
        Framework fw,
        String targetDevice,
        String objectName,
        String containerName,
        Object object,
        Object ticket)
```

© UC Regents, 2003

This method will store an object on the logical *targetDevice* managed by the storage manager. The *objectName* will specify a unique name (within the storage manager's name space) for the object. An optional *containerName* will provide a means for grouping the object with other objects in physical storage.

The *ticket* will provide proof of authorization to the storage manager. The ticket will be obtained through the Security function (see section 5.2.6.1.3.)

5.2.8.1.2 Method getObject

Signature:

```
public StorageResult getObject(
        Framework fw,
        String objectName,
        Object ticket)
```

This method will retrieve a binary object from storage. The *objectName* will specify the unique name of the object to be retrieved.

The *ticket* will provide proof of authorization to the storage manager. The ticket will be obtained through the Security function (see section 5.2.6.1.2)

5.2.8.1.3 Method removeObject

Signature:

```
public StorageResult removeObject(
        Framework fw,
        String objectName,
        Object ticket)
```

This method will remove a binary object from storage. The *objectName* will specify the unique name of the object to be removed.

The *ticket* will provide proof of authorization to the storage manager. The ticket will be obtained through the Security function (see section 5.2.6.1.2)

5.2.8.2 Class StorageManager_SRB

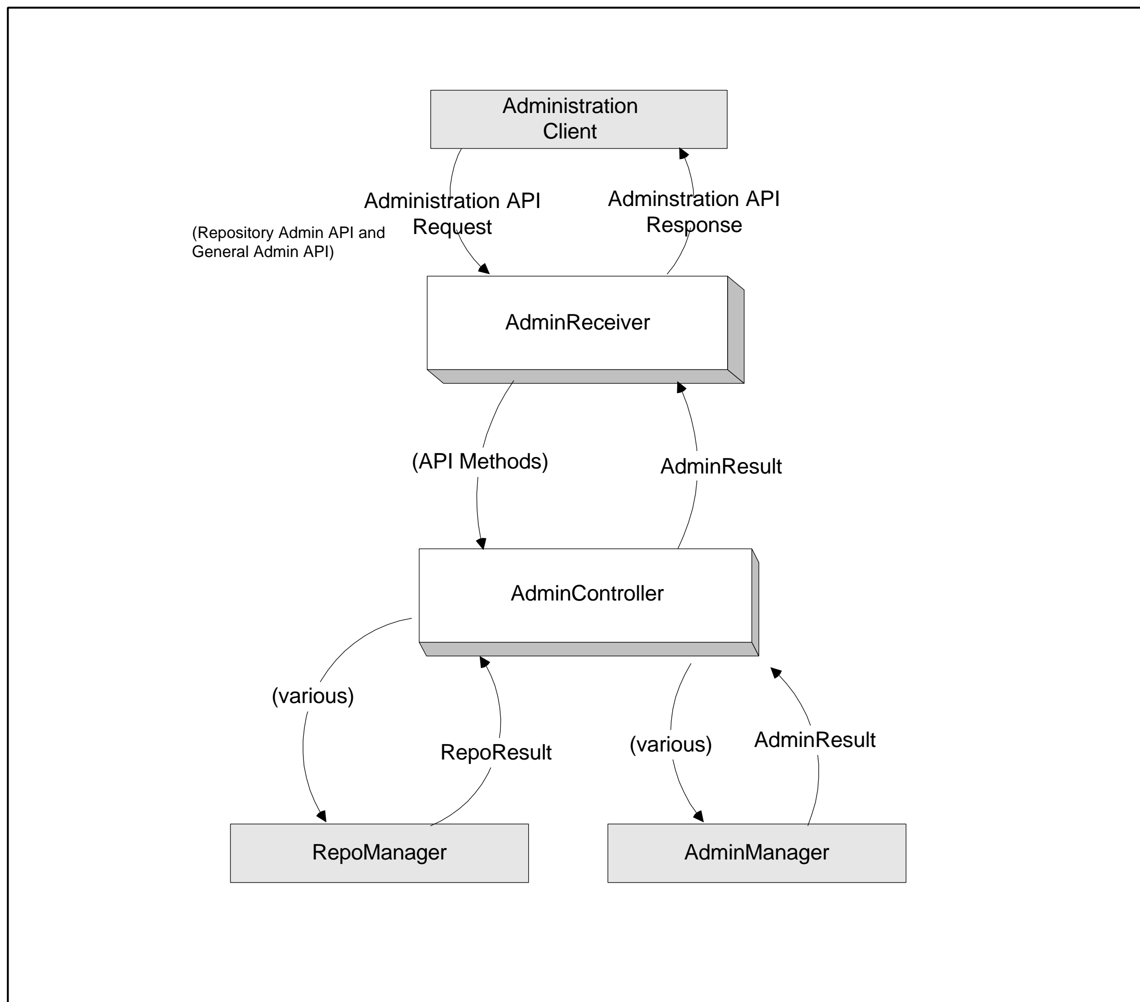
This class will implement the *StorageManager* interface using the Storage Resource Broker (SRB). It will utilize *JARGON*, a Java client API for SRB, to implement *StorageManager* methods as calls to SRB.

© UC Regents, 2003

5.2.9 Administration

The Administration component will provide administrators of the repository with functions to inventory, report on, and update the contents of the repository; and to manage data relating to producers, consumers, submission agreements, access rights and collections.

The function will receive Repository Administration API and General Administration API requests (see sections 5.4.3.1 and 5.4.3.2) coming from an Administration Client and translate the requests into actions for the Data Management component of the system. The following diagram illustrates the sub-components of Adminstration and the message flows among components and sub-components.



5.2.9.1 Class AdminReceiver

Class *AdminReceiver* will receive from an Admistration Client an HTTP-encapsulated stream of one or more SOAP-formatted Administration API requests (both Repository and General.)

The class will be implemented as a Common Framework component in package *org.cdlib.framework.admin*.

© UC Regents, 2003

5.2.9.1.1 Method init

Signature:

```
protected void init(ServletConfig config)
```

This method will create a new *Framework* (see section 5.2.11.1) object and store it as a member variable. Servlet parameters will provide the name(s) of properties file(s) that will be passed to the *Framework* constructor.

5.2.9.1.2 Method doPost

Signature:

```
protected void doPost(HttpServletRequest req,
                      HttpServletResponse resp)
```

This method will:

1. Process a stream containing messages conforming to request messages in the Repository Administrtion API specification (see section 5.4.3.1) or the General Administration API (section 5.4.3.2) The stream containing the API messages will be expected to be present in the body portion of the HTTP POST request.
2. For each request message in the stream, the method will invoke a method in *AdminController* that corresponds to the request message.

5.2.9.2 Class AdminController

Class AdminController will control the sequence of events necessary for processing an administration API request. A separate method will exist for each type of request defined in the Repository Administrtion API specification (see section 5.4.3.1) and the General Administration API (section 5.4.3.2.) Methods corresponding to Repository Administration API requests will return a *RepoResult* (see section 5.3.5.8) object; those that correspond to General Administration API requests will return an AdminResult (see section 5.3.5.9.) object.

The class will be implemented as a Common Framework component in package *org.cdlib.framework.admin*.

5.2.10 Administration Client

The Administration Client will interact with an administrator of the Preservation Repository and communicate with the Administration function (see section 5.2.9) to:

- Manage information about producers, consumers, submission agreements, collections and system access rights
- Produce inventories of objects belonging to producers, covered by a specific agreement, or included in a specific collection
- Produce reports of producers, consumers, submission agreements and collections

All communication between the Administration Client and the Administration component will be through the Repository Administration API (see section 5.4.3.1) and the General Administration API (section 5.4.3.2.)

© UC Regents, 2003

5.2.11 Infrastructure

The classes that implement the underlying infrastructure of the Common Framework (see section 5.1) are described in more detail in the following sub-sections.

5.2.11.1 Class Framework

The *Framework* class will be the keeper of all properties relating to the overall operation of the CF. A single instance of *Framework* will exist in each runtime environment. The following properties will be maintained:

- Component registry objects
- Logger object
- Global properties

The class will be implemented in package *org.cdlib.framework.utility*.

5.2.11.1.1 Constructor

Signature:

```
public Framework(String[] propertiesFileNames)
```

The constructor will initialize the Framework by reading one or more properties files into memory. It will then create a series of registry objects, passing each registry the properties list to enable registry of components based upon the contents of the list.

The following table summarizes the properties that register components:

Property	*Description*	*Example*
SIPParser.*format*	Implementation of *SIPParser* interface to be used for SIPS with the format '*format*'	**org.cdlib.framework.ingest.SIPParser_METS**
AIPValidator.*profile*	Implementation of *AIPValidator* interface to be used for profile '*profile*'	**org.cdlib.preservation.ingest.AIPValidator_BASIC**
AIPIngestor.*service*	Implementation of *AIPIngestor* interface to be used for service '*service*'	**org.cdlib.preservation.ingest.AIPIngestor_PRESERVE**
DIPRetriever.*service*	Implementation of *DIPRetriever* interface to be used for service '*service*'	**org.cdlib.preservation.access.DIPRetriever_PRESERVE**
DIPFormatter.*profile*	Implementation of *DIPFormatter* interface to be used for profile '*profile*'	**org.cdlib.preservation.access.DIPFormatter_BASIC**
SecurityManager.*service*	Implementation of *SecurityManager*	**org.cdlib.preservation.security.SecurityManager_PRESERVE**

© UC Regents, 2003

	interface to be used for service '*service*'	
RepoManager	Implementation of *RepoManager* to be used	**org.cdlib.framework.dataManagement.RepoManager_MySQL**
AdminManager	Implementation of *AdminManager* to be used	**org.cdlib.framework.dataManagement.AdminManager_MySQL**
IDManager	Implementation of *IDManager* to be used	**org.cdlib.framework.dataManagement.IDManager_ARK**
StorageManager	Implementation of *StorageManager* to be used	**org.cdlib.framework.archiveStorage.StorageManager_SRB**
Logger	Implementation of *Logger* to be used	**org.cdlib.framework.utility.FileLogger**

The constructor will also create and start a *Monitor* (see section 5.2.11.4) for the framework.

5.2.11.1.2 Method getSIPParser

Signature:

```
public SIPParser getSIPParser(String format)
```

This method will find the registered SIPParser class for format *format* and return an instance of the class.

5.2.11.1.3 Method getAIPValidator

Signature:

```
public AIPValidator getAIPValidator (String profile)
```

This method will find the registered AIPValidator class for profile *profile* and return an instance of the class.

5.2.11.1.4 Method getAIPIngestor

Signature:

```
public AIPIngestor getAIPIngestor(String service)
```

This method will find the registered AIPIngestor class for service *service* and return an instance of the class.

5.2.11.1.5 Method getDIPRetriever

Signature:

```
public DIPRetriever getDIPRetriever(String service)
```

This method will find the registered DIPRetriever class for service *service* and return an instance of the class.

5.2.11.1.6 Method getDIPFormatter

Signature:

```
public DIPFormatter getDIPFormatter(String profile)
```

© UC Regents, 2003

This method will find the registered DIPFormatter class for profile *profile* and return an instance of the class.

5.2.11.1.7 Method getSecurityManager

Signature:

```
public SecurityManager getSecurityManager(String service)
```

This method will find the registered SecurityManager class for service *service* and return an instance of the class.

5.2.11.1.8 Method getRepoManager

Signature:

```
public RepoManager getRepoManager()
```

This method will find the registered RepoManager implementation and return an instance of the class.

5.2.11.1.9 Method getAdminManager

Signature:

```
public AdminManager getAdminManager()
```

This method will find the registered RepoManager implementation and return an instance of the class.

5.2.11.1.10Method getIDManager

Signature:

```
public IDManager getIDManager()
```

This method will find the registered IDManager implementation and return an instance of the class.

5.2.11.1.11Method getStorageManager

Signature:

```
public StorageManager getStorageManager()
```

This method will find the registered StorageManager implementation and return an instance of the class.

5.2.11.1.12 Method getLogger

Signature:

```
public Logger getLogger()
```

This method will return the logger instance for the framework.

5.2.11.1.13 Method getProperty

Signature:

```
public String getProperty(String propertyName)
```

This method will return the value of a global property.

© UC Regents, 2003

5.2.11.1.14 Method getProperty

Signature:

```
public Monitor getMonitor()
```

This method will return the Monitor (see section 5.2.11.4) for the framework.

5.2.11.2 Interface Logger

This interface defines the behavior of a function for storing information and error messages.

5.2.11.2.1 Method logMessage

Signature:

```
public void logMessage(
        Framework fw,
        String message,
        int significance,
        Object source)
```

This method will store an information message in the log.

5.2.11.2.2 Method logError

Signature:

```
public void logError(
        Framework fw,
        String message,
        int significance,
        Object source)
```

This method will store an error message in the log.

5.2.11.3 Class FileLogger

This class will provide a basic implementation of a logger that writes to a file. It will implement interface *Logger.* The location of the file, the level of logging, and the time period corresponding to an instance of the file will be controlled by entries in the properties file (see section 5.2.11.1.1) as follows:

Property	*Description*	*Example*
fileLogger.message.maximumLevel	Maximum level of informational messages that will be logged (-1 – 10)	**1**
fileLogger.error.maximumLevel	Maximum level of error messages that will be logged (-1 – 10)	**10**
fileLogger.path	Path to directory for the log file	**/preservation/logs**
fileLogger.name	Name for log file	**preservation**
fileLogger.qualifier	Name qualifier that determines how often the log file is recycled.	**MMddyy**

Note that these properties will be retrieved via the *getProperty* method of the *Framework* class.

© UC Regents, 2003

5.2.11.4 Class Monitor

This class will monitor the operation of an instance of the framework and perform periodic maintenance.

5.2.11.4.1 Constructor

Signature:

```
public Monitor(Framework fw)
```

The constructor will start a thread to monitor the operation of the framework at a specified interval and to report errors via the *Logger* and electronic mail. Configuration information will be specified in the properties file (see section 5.2.11.1.1) as follows:

Property	*Description*	*Example*
monitor.intervalMS	Time between invocations of the monitor, in milliseconds	120000
monitor.alertRecipient	E-mail address for receiving alerts	**repo-l@cdlib.org**

Note that these properties will be retrieved via the *getProperty* method of the *Framework* class.

5.2.11.4.2 Method start

Signature:

```
public void start()
```

This method will start the monitor.

5.2.11.4.3 Method stop

Signature:

```
public void stop()
```

This method will stop the monitor.

5.3 Data Model

5.3.1 Submission Information Package (SIP)

The Submission Information Package will represent the information flowing from a producer into the Preservation Repository. This area of the model will be implemented entirely as Common Framework classes and interfaces in package *org.cdlib.framework.datamodel.*

© UC Regents, 2003

5.3.1.1 Class SIPackage

The *SIPackage* class will represent the information submitted by a producer through the Ingest API (see section 5.4.1.) It will contain the following properties:

- producerID – An identifier of the source of the SIP
- authCode – Authentification information
- serviceID – The service being requested. Will be 'PRESERVE' for the Preservation Repository
- agreementID – (Optional) An identifer of the submission agreement associated with the object. Will only be used if a new object is created.
- collectionID – (Optional) A collection identifier to be associated with the object. Will only be used if a new object is created.
- objectID – (Optional) The identifier (ARK name) of the object to which a version is to be added
- altObjectID – (Optional) A reference code used by the producer to identify the object in its local repository
- accessGroupID – (Optional) Identifier of the access group allowed to access the object. Default is 'ALL'. Will only be used if the object is created.
- objectFormat – Description of the digital object's metadata format (e.g., 'METS')
- object – The digital object's metadata in the format identified by *objectFormat*

5.3.2 Archival Information Package (AIP)

The Archival Information Package will represent content and metadata that has been normalized into the internally supported format of the Preservation Repository (and the Common Framework.) This area of the model will be implemented entirely as Common Framework classes and interfaces in package *org.cdlib.framework.datamodel.*

The structure of metadata in the AIP will adhere to the METS specification.

5.3.2.1 Class AIPackage

This class will hold all the information that will be required to store or revise a digital object in the preservation repository, including:

- metadata -- Metadata, represented as a "normalized" METS object hierarchy. The metadata will be represented as a Document Object Model (DOM) *Document* object.
- objectURIList – URI's for digital object(s) referenced by the metadata
- producerID – An identifier of the source of the submission
- altObjectID – An (optional) reference code used by the producer to identify the object in its local repository
- serviceID– An identification of the service ('PRESERVE' for the Preservation Repository)
- agreementID –An identifer of the submission agreement associated with the object.
- collectionID – (Optional) A collection identifier to be associated with the object.
- accessGroupID– Identifier of the access group permitted to access the object
- objectID – (Optional) The identifier (ARK name) of the object
- sipFormat – Format of the object metadata in the SIP from which the AIP was derived
- sip – Original contents of the object metadata in the SIP
- profile – The profile identifier for the object

© UC Regents, 2003

5.3.3 Dissemination Information Package (DIP)

The Dissemination Information Package will hold objects and metadata that have been retrieved from the repository. This area of the model will be implemented entirely as Common Framework classes and interfaces in package *org.cdlib.framework.datamodel.*

5.3.3.1 Class DIPackage

This abstract class will be the base class for classes that hold information retrieved from the repository for a stored digital object. It will include the following properties:

- profile – The profile identifier for the object

5.3.3.2 Class DIPackageFull

This class will extract and hold all information for a digital object that is stored in the repository. It will extend class *DIPackage*. It will include the following properties:

- metadata – All metadata, represented as a METS XML string
- objectList -- Digital object(s) referenced by the metadata

5.3.3.2.1 Constructors

Signatures:

```
public DIPackageFull(Framework fw, ObjectID objectID)
```

This constructor will create a *DIPackageFull* object from the current version of a digital object.

```
public DIPackageFull(Framework fw, ObjectID objectID, int version)
```

This constructor will create a *DIPackageFull* object from a specified version of a digital object.

5.3.3.3 Class DIPackageMeta

This class will extract and hold all metadata for a digital object that is stored in the repository. It will extend class *DIPackage*. It will include the following properties:

- metadata -- Metadata, represented as a METS XML string

5.3.3.3.1 Constructors

Signature:

```
public DIPackageMeta(Framework fw, ObjectID objectID)
```

This constructor will create a *DIPackageMeta* object from the current version of a digital object.

```
public DIPackageMeta (Framework fw, ObjectID objectID, int version)
```

This constructor will create a *DIPackageMeta* object from a specified version of a digital object.

5.3.3.4 Class DIPackageSummary

This class will extract and hold all summarized metadata for a digital object that is stored in the repository. It will extend class *DIPackage*. It will include the following properties:

© UC Regents, 2003

- metadata – Selective metadata, represented as a METS XML string

5.3.3.4.1 Constructors

Signature:

```
public DIPackageSummary(Framework fw, ObjectID objectID)
```

This constructor will create a *DIPackageSummary* object from the current version of a digital object.

```
public DIPackageSummary(Framework fw, ObjectID objectID, int version)
```

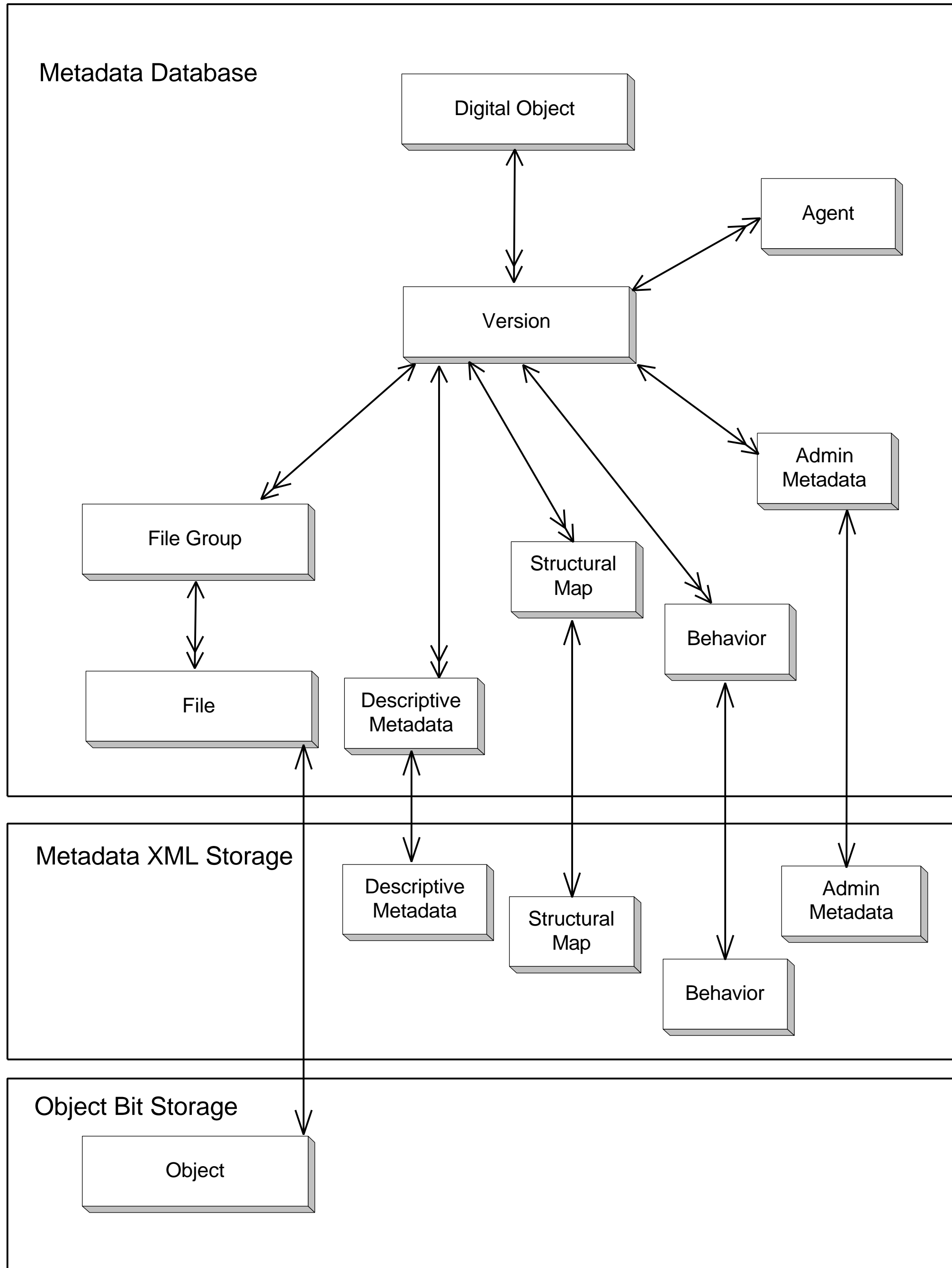
This constructor will create a *DIPackageSummary* object from a specified version of a digital object.

© UC Regents, 2003

5.3.4 Stored Data Model

5.3.4.1 Digital Object Repository Overview

The following diagram illustrates how a digital object will be represented in persistent storage.



As the diagrams shows, data will reside in three distinct persistent storage containers:

© UC Regents, 2003

- The *Metadata Database* will reside in a database management system's space. It will represent high-level metadata entities relating to the digital object and will maintain internal relationships among metadata entities and references to components residing in other storage containers.
- *Metadata XML Storage* will represent more detailed portions of metadata stored in the original XML text form of the submitted digital object. The XML will reside in files in a storage manager's space or in "blobs" in the database management system's space.
- *Object Bit Storage* will represent the digital content portion of the object in its original bit formation. It will reside in files in a storage manager's space.

The proposed logical structure of the digital object repository closely follows that of the Digital Library Federation's METS standard for XML encoding of digital object metadata: Database entities reflect the major METS document sections and database relationships reflect METS section relationships (nesting and repetition.)

The database will contain a separate entity, *Service Metadata,* to house control information attached to a digital object by the Preservation Repository's ingest process.

Some portions of metadata will be represented in both the database and the Metadata Bit Storage area. High level, structured metadata – parsed during ingest of the SIP – will reside in the database; complex, low level "raw" metadata will reside in the bit storage area. The database entity will maintain a reference to the raw data.

The content portion(s) of a digital object will reside in the Object Bit Storage area. *File* entities in the database will maintain references to the content.

5.3.4.2 Metadata Database – MySQL Implementation

The following sections describe an implementation of the metadata database using MySQL. In this implementation, a series of tables will be defined in a Repository database. Metadata XML will be stored in blobs in the tables.

5.3.4.2.1 Table DO_DIGITAL_OBJECT

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
producerID	index (see note)	Producer of the object
altObjectID	index (see note)	An alternative code used by the producer to identify the object
agreementID	index	Submission agreement identifier
collectionID	index	Collection identifier
serviceID	index	Service identifier
accessGroupID		Access Group permitted to access the object
creationDate		Date object was created
currentVersionID		Current version

Note: Column *producerID* is a simple index; columns *producerID* and *altObjectID* together form a composite index.

5.3.4.2.2 Table DO_VERSION

This table corresponds to the *mets* element of the METS schema, with the addition of a version ID and version date.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier

© UC Regents, 2003

versionDate		Date version was added to the object
label		Title/text string identifying the object
type		The type of the object, e.g., book, journal
profile		A profile to which the SIP form of the object conformed

5.3.4.2.3 Table DO_AGENT

This table corresponds to an *agent* element and and a corresponding *name* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
agentID	key	Unique ID for the agent (system assigned)
role		Agent's role with respect to the object (enumeration; see METS schema)

5.3.4.2.4 Table DO_FILEGROUP

This table corresponds to the *fileGrp* element of the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
fileGroupID	key	Identifier for the file group; optional; default is '0'
parentGroupID		Identifier of parent file group (enables recursion)

5.3.4.2.5 Table DO_FILE

This table corresponds to a *file* element and an embedded *FLocat* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
fileGroupID	key	Identifier for the file group; optional; default is '0'
fileID	key	Identifier for the file
storageID	index	The identifier assigned to the file in archival storage (enables retrieval of the physical file)
mimeType		Mime type of the file's contents

5.3.4.2.6 Table DO_STRUCTMAP

This table corresponds to a *structMap* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
structureID	key	Identifier for the structural map metadata
type		Type of structural map
label		Short description of the structural map
metadata	(blob)	Metadata (*div* elements embedded in structural map)

© UC Regents, 2003

5.3.4.2.7 Table DO_DESCRIPT_MD

This table corresponds to a *dmdSec* element and an embedded *mdwrap* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
descriptID	key	Identifier for the descriptive metadata
mdType		Type of descriptive metadata (enumeration; see METS schema)
label		Short description of the metadata
metadata	(blob)	Metadata

5.3.4.2.8 Table DO_RIGHTS_MD

This table corresponds to a *rightsMD* element and an embedded *mdwrap* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
rightsID	key	Identifier for the rights metadata
label		Short description of the metadata
metadata	(blob)	Metadata

5.3.4.2.9 Table DO_TECH_MD

This table corresponds to a *techMD* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
techID	key	Identifier for the technical metadata
metadata	(blob)	Metadata

5.3.4.2.10 Table DO_SOURCE_MD

This table corresponds to a *sourceMD* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
sourceID	key	Identifier for the source metadata
metadata	(blob)	Metadata

5.3.4.2.11 Table DO_DIGIPROV_MD

This table corresponds to a *sourceMD* element in the METS schema.

Column	*Key/Index*	*Description*
objectID	key	Object Identifier (ARK)
versionID	key	Version identifier
digiProvID	key	Identifier for the digital provenance metadata
metadata	(blob)	Metadata

5.3.4.2.12 Table DO_BEHAVIOR_MD

This table corresponds to a *behavior* element in the METS schema.

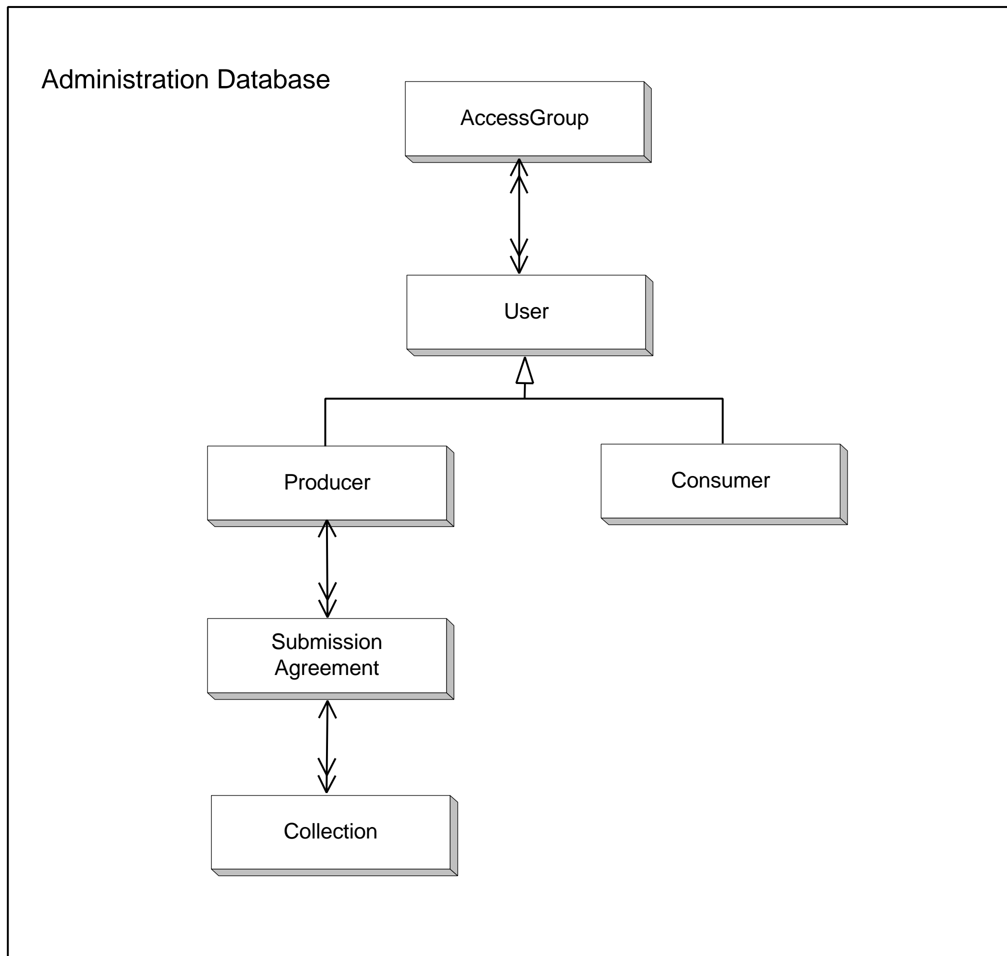
© UC Regents, 2003

Column	*Key/Index*	*Description*
`objectID`	key	Object Identifier (ARK)
`versionID`	key	Version identifier
`behaviorID`	key	Identifier for the behavior metadata
`btype`		Type of behavior (e.g., display)
`structID`		Reference to structure map that is the target of the behavior
`label`		Short description of the behavior
`metadata`	(blob)	Metadata

5.3.4.3 Administration Database

The Administration Database will store information for defining and describing producers, consumers, submission agreements, object collections, access groups and other entities relating to the management of the repository.

The following diagram illustrates these entities and their relationships:



- The *User* entity defines a user of the respository, which can be either a *Producer* or a *Consumer*
- The *Producer* entity defines a source of digital objects.
- The *Consumer* entity defines a party that accesses digital objects stored in the repository.

© UC Regents, 2003

- An *Access Group* connects *Producers* to *Consumers* and serves to identify *Consumers* that are permitted by a *Producer* to view a group of objects. An *Access Group* may contain more than one *Producer* and more than one *Consumer*; and *Producers* and *Consumers* can belong to more than one *Access Group*.
- A *Submission Agreement* is a formal contract executed by the UC Libraries and a *Producer* that authorizes the *Producer* to submit a body of digital objects. A *Producer* may execute more than one *Submission Agreement*.
- A *Collection* is a sub-grouping of digital objects submitted under a single *Submission Agreement*.

5.3.4.4 Administration Database – MySQL Implementation

The following sections describe an implementation of the Administration Database using MySQL. In this implementation, administration tables will be included in the Repository Database (see section 5.3.4.2).

5.3.4.4.1 Table AD_ACCESS_GROUP

Column	*Key/Index*	*Description*
accessGroupID	key	Group identification
name		Name of access group

5.3.4.4.2 Table AD_PRODUCER

Column	*Key/Index*	*Description*
producerID	key	Producer identification (use MARC Organization Code?)
name	index	Name of producer organization
address1		Producer’s street address (part 1)
address2		Producer’s street address (part 2)
city		Producer’s city
state		Producer’s state or province
postalCode		Producer’s postal code
country		Producer’s country
contactName		Name of contact person
contactTitle		Title of contact person
contactEmail		Email Address of contact person
contactPhone		Phone number of contact person

5.3.4.4.3 Table AD_CONSUMER

Column	*Key/Index*	*Description*
consumerID	key	Consumer identification
name	index	Name of consumer organization

5.3.4.4.4 Table AD_USER

Column	*Key/Index*	*Description*

© UC Regents, 2003

userID	key	User identification
authCode	(encrypted)	Authorization code
userType		'P' for producer; 'C' for consumer

5.3.4.4.5 Table AD_ACCESS_GRP_MEMBER

Column	*Key/Index*	*Description*
accessGroupID	key	Group identification
userID	key	User identification

5.3.4.4.6 Table AD_SUBMISSION

Column	*Key/Index*	*Description*
agreementID	key	Submission identifier
producerID	index	Identifier of producer organization
description		Description of submission agreement
contactName		Name of contact person (for submission agreement)
contactTitle		Title of contact person
contactEmail		Email Address of contact person
contactPhone		Phone number of contact person

5.3.4.4.7 Table AD_COLLECTION

Column	*Key/Index*	*Description*
collectionID	key	Collection identifier
agreementID	index	Submission agreement identifier
name		Collection name
description		Collection description

5.3.5 Control Information

This area of the PR data model will comprise classes that provide system control information and facilitate communication between functions. It will be implemented as Common Framework classes in package *org.cdlib.framework.datamodel.control.*

5.3.5.1 Class ObjectID

Class ObjectID will represent a unique identifier for a digitial object.

5.3.5.1.1 Constructor

Signature:

```
public ObjectID(String id, String producerID, String serviceID)
```

© UC Regents, 2003

5.3.5.2 Class Request

Class *Request* will hold properties describing a request made to the PR by a producer, consumer or other external entity and will be the mechanism for determining if the requestor is allowed to perform an action.

Properties:

- requestorID – Identifier of the entity making the request
- authCode – Authentication information provided by the requestor
- service – The service area of the request (always "PRESERVE" for PR)
- action – The action being requested within the service area
- ticket – If a requestor has been issued an authorization ticket, the ticket that was issued

5.3.5.2.1 Constructor

Signature:

```
publc Request(String requestorID, String authCode, String service,
String action)
```

5.3.5.2.2 Method isAllowed

Signature:

```
publc Boolean isAllowed()
```

This method will determine if the requestor is authorized to perform the action specified in the service and action properties. It will determine this via the Security component (see section 5.2.4.4.2.)

5.3.5.3 Class Query

Class *Query* will hold properties describing a query submitted by a consumer or administrator.

Properties:

- queryText – Text of query to be executed

5.3.5.3.1 Constructor

Signature:

```
public Query(String queryText)
```

5.3.5.4 Class IngestResult

This class will hold properties that communicate the results of operations relating to Ingest (see section 5.2.2.)

Properties:

- statusCode – Code indicating the success or cause of failure of a process
- statusMessage – Text associated with *statusCode*
- objectID – Object identifier resulting from a process

© UC Regents, 2003

5.3.5.4.1 Constructor

Signature:

```
public IngestResult(int statusCode, String statusMessage, ObjectID
objectID)
```

5.3.5.5 Class AccessResult

This class will hold properties that communicate the results of operations relating to Access (see section 5.2.4.)

Properties:

- statusCode – Code indicating the success or cause of failure of a process
- statusMessage – Text associated with *statusCode*
- dip – DIPackage (see section 5.3.3.1) resulting from a successful access process

5.3.5.5.1 Constructor

Signature:

```
public AccessResult(int statusCode, String statusMessage, DIPackage dip)
```

5.3.5.6 Class SIPParseResult

This class will hold properties that communicate the results of a SIP parsing process (see section 5.2.3.4.)

Properties:

- statusCode – Code indicating the success or cause of failure the parsing process
- statusMessage – Text associated with *statusCode*
- aip – AIPackage (see section 5.3.2.1) resulting from a successful parsing process

5.3.5.6.1 Constructor

Signature:

```
public SIPParseResult(
        int statusCode, String statusMessage, AIPackage aip)
```

5.3.5.7 Class SecurityResult

This class will hold properties that communicate the results of operations relating to Security (see section 5.2.5.)

Properties:

- statusCode – Code indicating the success or cause of failure of a security check
- statusMessage – Text associated with *statusCode*
- ticket – Ticket resulting from the security check

5.3.5.7.1 Constructor

Signature:

© UC Regents, 2003

```
public SecurityResult(
        int statusCode, String statusMessage, Object ticket)
```

5.3.5.8 Class RepoResult

This class will hold properties that communicate the results of a RepoManager process (see section 5.2.7.1.)

Properties:

- statusCode – Code indicating the success or cause of failure the repository management process
- statusMessage – Text associated with *statusCode*
- dipArray – Array of DIPackages (see section 5.3.3) resulting from a retrieval process

5.3.5.8.1 Constructor

Signature:

```
public RepoResult(
        int statusCode, String statusMessage, DIPackage[] dip)
```

5.3.5.9 Class AdminResult

This class will hold properties that communicate the results of an AdminManager process (see section 5.2.7.3.)

Properties:

- statusCode – Code indicating the success or cause of failure the repository management process
- statusMessage – Text associated with *statusCode*
- resultSet – ResultSet resulting from a retrieval process

5.3.5.9.1 Constructor

Signature:

```
public AdminResult(
        int statusCode, String statusMessage, ResultSet resultSet)
```

5.3.5.10 Class StorageResult

This class will hold properties that communicate the results of a StorageManager process (see section 5.2.8.1.)

Properties:

- statusCode – Code indicating the success or cause of failure the storage management process
- statusMessage – Text associated with *statusCode*
- object – Object retrieved (when applicable)

5.3.5.10.1 Constructor

Signature:

```
public StorageResult(
```

© UC Regents, 2003

```
int statusCode, String statusMessage, Object object)
```

5.4 Interfaces

5.4.1 Ingest API

The Ingest Application Program Interface (Ingest API) will enable Ingest client programs to establish objects in the repository, submit new versions of objects for archiving and to update or delete previously submitted versions of objects. The API will be implemented as a web service adhering to the Simple Object Access Protocol (SOAP) specification.

5.4.1.1 Method createObject

This method will generate an object identifier (ARK name) and establish a digital object in the repository.

Parameters:

- producerID – An identifier of the source of the request
- authCode – Authentification information
- serviceID – The service being requested. Will be 'PRESERVE' for the Preservation Repository
- agreementID – (Optional) An identifer of the submission agreement associated with the object being created
- collectionID – (Optional) A collection identifier to be associated with the object
- altObjectID – (Optional) A reference code used by the producer to identify the object in its local repository
- accessGroupID – (Optional) Identifier of the access group allowed to access the object. Default is 'ALL'.

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- objectID – The object ID (ARK name) assigned to the created object
- altObjectID – The producer's object reference, echoed back from the request parameter.

5.4.1.2 Method addVersion

This API method will add a new version of a digital object and set the digital object's *current version* to the new version.

The digital object will be located by specifying its identifier in the *objectID* parameter and/or by specifying an alternative identifier in the *altObjectID* parameter. If both parameters are present, both must match an existing object. If only the *altObjectID* is present and it does not match an existing object, or if neither parameter is present, a new object will be created and a version added to it. Note that *producerID* is used to qualify *altObjectID* in performing matching.

Parameters:

- producerID – An identifier of the source of the request
- authCode – Authentification information
- serviceID – The service being requested. Will be 'PRESERVE' for the Preservation Repository

© UC Regents, 2003

- agreementID – (Optional) An identifer of the submission agreement associated with the object. Will only be used if a new object is created.
- collectionID – (Optional) A collection identifier to be associated with the object. Will only be used if a new object is created.
- objectID – (Optional) The identifier (ARK name) of the object to which a version is to be added
- altObjectID – (Optional) A reference code used by the producer to identify the object in its local repository
- accessGroupID – (Optional) Identifier of the access group allowed to access the object. Default is 'ALL'. Will only be used if the object is created.
- objectFormat – Description of the digital object's metadata format (e.g., 'METS')
- object – The digital object's metadata (base64 encoded) in the format identified by *objectFormat*

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- objectID – The public identifier (ARK name) of the object to which the version was added. This element will only be present in the response if the version was added successfully
- altObjectID – The producer's object reference, echoed back from the request parameter.

5.4.1.3 Method replaceVersion

This API method will replace the current version of a digital object in the repository. Object matching rules will be identical to those for the *addVersion* method.

If the matching object does not have a current version, a new version will be added. If there is no matching object, a new object will be created and a version added to it.

Parameters:

- producerID – An identifier of the source of the request
- authCode – Authentification information
- serviceID – The service being requested. Will be 'PRESERVE' for the Preservation Repository
- agreementID – (Optional) An identifer of the submission agreement associated with the object. Will only be used if a new object is created.
- collectionID – (Optional) A collection identifier to be associated with the object. Will only be used if a new object is created.
- objectID – (Optional) The identifier (ARK name) of the object for which a version is to be replaced
- altObjectID – (Optional) A reference code used by the producer to identify the object in its local repository
- accessGroupID – (Optional) Identifier of the access group allowed to access the object. Default is 'ALL'. Will only be used if the object is created.
- objectFormat – Description of the digital object's metadata format (e.g., 'METS')
- object – The digital object's metadata (base64 encoded) in the format identified by *objectFormat*

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- objectID – The public identifier (ARK name) of the object for which the version was replaced. This element will only be present in the response if the version was replaced successfully
- altObjectID – The producer's object reference, echoed back from the request parameter.

© UC Regents, 2003

5.4.1.4 Method removeVersion

This API method will remove the current version of a digital object from the repository and set the digital object's current version to the previous version, if one exists. Object matching rules will be identical to those for the *addVersion* method.

Parameters:

- producerID – An identifier of the source of the request
- authCode – Authentification information
- serviceID– The service being requested. Will be 'PRESERVE' for the Preservation Repository
- objectID – (Optional) The identifier (ARK name) of the object for which a version is to be removed
- altObjectID – (Optional) A reference code used by the producer to identify the object in its local repository

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- altObjectID – The producer's object reference, echoed back from the request parameter.

5.4.1.5 Method validateObject

This API method will determine if a digital object is acceptable to the Preservation Repository and could therefore be submitted to the repository. This provides a producer with a means of validating objects before attempting to submit them in an addVersion or replaceVersion request.

Parameters:

- producerID – An identifier of the source of the submission
- authCode – Authentication information
- serviceID– The service being requested. Will be 'PRESERVE' for the Preservation Repository
- altObjectID – (Optional) A reference code used by the producer to identify the object in its local repository
- objectFormat – Description of the digital object's metadata format (e.g., 'METS')
- object – The digital object's metadata (base64 encoded) in the format identified by *objectFormat*

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- altObjectID – The producer's object reference, echoed back from the request parameter.

5.4.2 Access API

The Access Application Program Interface (Access API) will enable Access client programs to retrieve digital objects from the repository. The API will be exposed as a web service adhering to the Simple Object Access Protocol (SOAP) specification.

5.4.2.1 Method getObject

This API method will retrieve a digital object from the Preservation Repository.

© UC Regents, 2003

Parameters:

- consumerID – An identifier of the source of the access request
- authCode – Authentification information
- service – An identifier of the service that holds the object. For the Preservation Repository, this will be 'PRESERVE'
- objectID – Identifier (awk name) of the object to be retrieved
- version – The version of the object to retrieve. This parameter is optional; if it is not present, the current version of the object will be retrieved.

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- dip – Serialized *DIPackageFull* for the requested object

5.4.3 Administration API

The Administration API will enable administrators of the repository to manage digital objects in the repository and to manage administrative information about producers, consumers, submission agreements and defined collections.

5.4.3.1 Repository Administration API

5.4.3.1.1 Method getObjectMetadata

This API method will retrieve the metadata for a digital object from the repository.

Parameters:

- consumerID – An identifier of the source of the access request
- authCode – Authentification information
- service – An identifier of the service that holds the object. For the Preservation Repository, this will be 'PRESERVE'
- objectID – Identifier (awk name) of the object whose metadata is to be retrieved
- version – The version of the metadata to retrieve. This parameter is optional; if it is not present, the current version of the metadata will be retrieved.

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- dip – Serialized *DIPackageMeta* for the requested object (see section 5.3.3.3)

5.4.3.1.2 Method getObjectSummary

This API method will retrieve partial metadata for a digital object from the repository.

Parameters:

- consumerID – An identifier of the source of the access request
- authCode – Authentification information

© UC Regents, 2003

- service – An identifier of the service that holds the object. For the Preservation Repository, this will be 'PRESERVE'
- objectID – Identifier (awk name) of the object whose partial metadata is to be retrieved
- version – The version of the metadata to retrieve. This parameter is optional; if it is not present, the current version of the metadata will be retrieved.

Returns:

- statusCode – Code indicating success or type of failure
- statusMessage – Text associated with the code
- dip – Serialized *DIPackageMeta* for the requested object (see section 5.3.3.3)

© UC Regents, 2003

5.4.3.2 General Administration API

5.4.3.2.1 Method createProducer

5.4.3.2.2 Method updateProducer

5.4.3.2.3 Method removeProducer

5.4.3.2.4 Method getProducer

5.4.3.2.5 Method getProducerList

5.4.3.2.6 Method createConsumer

5.4.3.2.7 Method updateConsumer

5.4.3.2.8 Method removeConsumer

5.4.3.2.9 Method getConsumer

5.4.3.2.10 Method getConsumerList

5.4.3.2.11 Method createAccessGroup

5.4.3.2.12 Method updateAccessGroup

5.4.3.2.13 Method removeAccessGroup

5.4.3.2.14 Method getAccessGroup

5.4.3.2.15 Method getAccessGroupList

5.4.3.2.16 Method addAccessGroupMember

5.4.3.2.17 Method removeAccessGroupMember

5.4.3.2.18 Method getAccessGroupMemberList

5.4.3.2.19 Method getSubmission

5.4.3.2.20 Method createSubmission

5.4.3.2.21 Method updateSubmission

5.4.3.2.22 Method removeSubmission

5.4.3.2.23 Method getSubmissionList

5.4.3.2.24 Method getSubmission

5.4.3.2.25 Method getCollection

5.4.3.2.26 Method createCollection

5.4.3.2.27 Method updateCollection

5.4.3.2.28 Method removeCollection

5.4.3.2.29 Method getCollectionList

© UC Regents, 2003

5.4.3.2.30 Method getCollection

© UC Regents, 2003