



Project no. 033572

CASPAR

Cultural, **A**rtistic and **S**cientific knowledge for **P**reservation, **A**ccess and **R**etrieval

Instrument: Information Society Technologies

Thematic Priority: 2.5.10 Access to and preservation of cultural and scientific resources

D2201: PRESERVATION DATASTORES INTERFACE



Document identifier:	CASPAR-D2201-TN-0101-1_1
Submission Date:	30-11-2008 (V 1.1)
Due date:	31-12-2007 (V 1.0)
Work package:	2200
Partners:	All Partners
WP Lead Partner:	IBM
Document status	DRAFT

Abstract: This document provides the first version of the Preservation DataStores interface.



Project information

Project acronym:	CASPAR
Project full title:	Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval
Proposal/Contract no.:	IST-2006-033572

Project Officer: Carlos Oliveira

Address:	<p>INFSO-E3 Information Society and Media Directorate General Content - Learning and Cultural Heritage</p> <p>Postal mail: Bâtiment Jean Monnet (EUFO 1167) Rue Alcide De Gasperi / L-2920 Luxembourg</p> <p>Office address: EUROFORUM Building - EUFO 1167 10, rue Robert Stumper / L-2557 Gasperich / Luxembourg</p>
Phone:	+352 4301 33052
Fax:	+352 4301 33190
Mobile:	
E-mail:	Carlos.Oliveira@ec.europa.eu

Project Co-ordinator: David Giaretta

Address:	<p>STFC (formerly CCLRC), Rutherford Appleton Laboratory</p> <p>Chilton, Didcot, Oxon OX11 0QX, UK</p>
Phone:	+44 1235 446235
Fax:	+44 1235 446362
Mobile:	+44 (0) 7770326304
E-mail:	d.i.giaretta@rl.ac.uk





CONTENT

1	INTRODUCTION.....	6
1.1	PURPOSE OF THIS DOCUMENT.....	6
1.2	HOW TO READ THIS DOCUMENT.....	6
1.3	GLOSSARY	6
2	BACKGROUND.....	8
2.1	BIT AND LOGICAL PRESERVATION	8
2.2	OAIS.....	9
2.3	PRESERVATION DATASTORES (PDS) WITHIN CASPAR	10
3	PRESERVATION AWARE STORAGE.....	12
3.1	MOTIVATION.....	12
3.2	REQUIREMENTS.....	12
4	PRESERVATION DATASTORES (PDS) ARCHITECTURE.....	14
4.1	ARCHITECTURE OVERVIEW	14
4.2	PRESERVATION ENGINE LAYER	15
4.2.1	Managing availability/data loss.....	16
4.2.2	AIP transformations.....	16
4.2.3	AIP identifier generation.....	16
4.2.4	Storlet container.....	17
4.2.5	Manage preservation specific metadata.....	17
4.3	XAM LAYER	18
4.4	OBJECT LAYER	19
5	INTEGRATING PDS WITH SRB/IRODS AND EXISTING ARCHIVES.....	21
5.1	PDS AND SRB/IRODS.....	21
5.2	PDS AND EXISTING ARCHIVES.....	22
6	AIP REPRESENTATION IN PDS AND MAPPING.....	24
6.1	AIP REPRESENTATION IN PRESERVATION ENGINE	24
6.2	PRESERVATION ENGINE MAPPING TO XAM	25
6.3	XAM MAPPING TO OSD.....	26
7	PDS INTERFACES.....	27
7.1	INTERFACE OVERVIEW.....	27
7.2	PDSMANAGER INTERFACE	28
7.3	PDSINTEGRATEDMANAGER INTERFACE.....	30
7.4	PDSMIGRATIONMANAGER INTERFACE	30
7.5	PDSPDiMANAGER INTERFACE.....	30
7.6	PDSREPIFOMANAGER INTERFACE	30
7.7	PDSPACKAGINGMANAGER INTERFACE	31
7.8	PDSAIP INTERFACE	31
7.9	PDSPDI INTERFACE.....	32
7.10	PDSPDIRECORD INTERFACE	32
7.11	PDSREPIFO INTERFACE.....	33
7.12	PDSREPIFORECORD INTERFACE.....	33
7.13	PDSAIPID INTERFACE.....	33
7.14	PDSAIPPOLICY INTERFACE.....	34
7.15	PDSPOLICY INTERFACE.....	35
7.16	PDSLINK INTERFACES.....	35
7.17	PDSPACKAGINGHANDLER INTERFACES	36
8	DATA FLOW	38
8.1	INGEST AIP.....	38
8.2	ACCESS AIP	38





8.3	MIGRATE AIP.....	39
9	AIP EXAMPLE: MAPPING A SAFE MANIFEST FILE TO AIP	40
9.1	GENERAL DESCRIPTION	40
9.2	SUGGESTED MAPPING	40
10	XFDU AIP GENERATOR.....	44
10.1	ADVANTAGES OF XAG.....	44
10.2	INSTRUCTIONS FOR USE	44
11	REFERENCES.....	47
	APPENDIX A – SAFE PRODUCT MANIFEST FILE	48

FIGURES

<i>Figure 1: OAI AIP logical structure</i>	9
<i>Figure 2: CASPAR System</i>	10
<i>Figure 3: Preservation DataStore architecture</i>	15
<i>Figure 4: SRB system</i>	21
<i>Figure 5: PDS and SRB/iRODS</i>	22
<i>Figure 6: Integrating PDS with existing archives or file systems</i>	23
<i>Figure 7: Representation of AIP in Preservation Engine layer</i>	24
<i>Figure 8: Flat mapping of AIP to XSet</i>	25
<i>Figure 9: PDS interfaces.....</i>	28
<i>Figure 10: PDSManager interface.....</i>	29
<i>Figure 11: PDSIntegratedManager interface</i>	30
<i>Figure 12: PDSMigrationManager interface</i>	30
<i>Figure 13: PDSPdiManager interface</i>	31
<i>Figure 14: PDSRepInfoManager interface</i>	31
<i>Figure 15: PDSPackagingManager interface.....</i>	31
<i>Figure 16: PDSAip interface and its related interfaces</i>	32
<i>Figure 17: PDSAipId interface</i>	33
<i>Figure 18: PDSAipPolicy interface.....</i>	34
<i>Figure 19: PDSLink interface and sub-interfaces.....</i>	36
<i>Figure 20: PDSPackagingHandler and PDSPackagingHandlerCreator interfaces</i>	36
<i>Figure 21: XAG overview.....</i>	45
<i>Figure 22: Addition menu with metadata section available</i>	46
<i>Figure 23: Addition menu with metadata section grayed out</i>	46

TABLES

Table 1: Glossary	6
Table 2: PDS manager interfaces summary	27
Table 3: Mapping SAFE manifest file to AIP	40





1 INTRODUCTION

1.1 PURPOSE OF THIS DOCUMENT

The purpose of this document is to bring an overview and describe the external interface of Preservation DataStores (PDS) which serves as the storage component of the CASPAR project. The overview includes the basic concepts of PDS along with PDS architecture and functionality. Then, we provide a detailed description of the PDS interfaces.

This document enables the understanding of the features and the usability of the PDS component. The PDS interfaces definition is iterative, and might be updated as we move to the next PDS phases.

1.2 HOW TO READ THIS DOCUMENT

The first part of the document which is composed of sections 1-6, describes the PDS system. It begins with a background followed by an overview of a new concept we developed called Preservation Aware Storage. Then, we present PDS architecture and describe its various layers. Next, we describe the integration of PDS with existing prominent archive technologies such as SRB/iRODS as well as suggest how to support data that already resides in existing file systems and archives. Finally, we conclude this part with a section that explains the representation and mapping of AIPs in the PDS various layers. The content of this part of the document is taken from our publications [1, 2, 3].

The second part of the document which is composed of sections 7-8, describes the PDS interfaces in details. It includes an overview of the PDS interfaces accompanied with UML diagrams and documentation to give further information about the main interfaces. We also present the PDS data flow.

The third part of the document which is composed of sections 9-10, demonstrates PDS with a specific type of scientific data. We show how this data is mapped in PDS and describe a tool to assist in generating AIPs for that data.

1.3 GLOSSARY

Table 1: Glossary

Bit preservation	The processes used to ensure that the bits comprising a preserved object are not lost or become corrupted over time. These processes include refreshing, backups, and error correcting code modules.
CASPAR	Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval
Content data object	The 'raw' data that makes up the content to be preserved
Content information object	The raw data and the metadata needed to interpret it; namely the content data plus its representation information
Context	The relationship of the content information to its environment, initially as perceived by the content data object provider. Later on, more context information may be implied or derived from the preservation process
Designated community	The primary OAIS user group that needs to access and understand the information preserved in the OAIS system. This means that the OAIS must have an appreciation of the community's knowledge base.





Digital preservation	The series of managed activities necessary to ensure continued access to digital materials for as long as necessary.
Fixity	The information that documents the authentication mechanisms and provides authentication keys to ensure that the content information object has not been altered in an undocumented manner.
Logical preservation	The processes used to ensure the understandability and usability of the data, in spite of the unknown changes in technologies and users in the future.
Metadata	Information about the content data object that is needed for its preservation.
Migration	The act of moving data from one system to another because of a change.
OAIS	Open Archival Information System. An ISO standard (ISO 14721:2003 OAIS) that specifies a reference model for an archive, consisting of an organization of people and systems that have accepted the responsibility to preserve information for a designated community.
Preservation system	An archiving system in which the lifetime of the data it needs to store exceeds the lifetime of the program/format with which the data is interpreted and the lifetime of the media that stores the bits.
Preservation aware storage	The storage component of a digital preservation system that has built-in support for preservation.
Preservation DataStores (PDS)	A new OAIS-based preservation aware storage. It will also serve as the storage component of CASPAR infrastructure.
Provenance	The information that documents the history of the content information. This information describes the origin or source of the content information, any changes that may have taken place since it was originated, and who has had custody of it since it was originated.
Representation information	The information that is required to interpret the content data object (raw data) into more meaningful concepts (ultimately more meaningful for humans).





2 BACKGROUND

The growth of long-lived digital information, along with new compliance regulations such as HIPAA, Sarbanes-Oxley, OSHA and other federal securities laws and regulations, demand the long-term viability of data. Other types of data must be preserved for the benefit of humankind. Just some examples include earth observation data from the European space agency and cultural heritage data from UNESCO, which must be kept for decades, centuries, or longer. Additionally, the amount of long-lived data is expected to grow as more digital devices generate vast amounts of born-digital data. This increases the need for digital preservation systems to preserve a myriad of information types including scientific, financial, healthcare, artistic, and cultural data—for tens and hundreds of years. Most of this information is reference data; it hardly changes once written. Due to its nature, this kind of data is typically accessed infrequently. Consequently, preservation systems generally utilize near-line and offline storage.

2.1 BIT AND LOGICAL PRESERVATION

Digital Preservation includes two related technologies, “bit preservation” and “logical preservation” (also called information preservation). Bit preservation includes the ability to restore the bits in the presence of storage media degradation and obsolescence, or even environmental catastrophes like fire and flooding. For example, bit preservation is responsible for ensuring the bits can be read after 5 years. A more complex example for bit preservation is ensuring the restoration of bits that were created on a 5¼” floppy disk located in a storage room that was damaged by fire.

Logical preservation includes preserving the understandability and usability of the data, despite unknown future changes that will take place in technologies and users (designated community). The data needs to be properly accessed and interpreted in the far future when current technologies for servers, operating systems and data management products may no longer exist. For example, assuming we can get the bits of a Word 3.0 document, how do we now read, understand and interpret it? Additionally, the logical preservation needs to maintain the provenance of the data, its authenticity, its integrity, and ensure that only legitimate users will access it.

Bit preservation is typically the responsibility of the storage layer. The storage layer includes data protection mechanisms such as RAID, or erasure coding. It may include mirroring services that maintain at least two copies of each data object, with each copy held at a physically separate location. Because the value of the data objects is mixed, the degree of protection for each data object may vary according to its importance. Because the data is located in multiple places, replicated data may introduce difficulties in refreshing, migrating, versioning, and access control.

Moreover, media may deteriorate. To support bit preservation, the storage layer refreshes the data according to a refreshment schedule. Generally, this is done every three years for hard disks and every five years for tapes. Although the life expectancy of tape is 10 to 30 years, refreshment is done sooner because of tape technology updates. Bit preservation strategies are well known and have been well tested in applied information technology. The challenge is mostly organizational rather than technical.

In contrast, logical preservation is still an unresolved problem. It is a recursive problem, where in addition to storing the raw data, it must also store the separately-born (in time and place) metadata that helps interpret and use the data itself. Moreover, this metadata (representation information) may need recursively additional metadata to help interpret it. The recursion ends when the representation information is non-digital and preserved by the designated community.

To further support logical preservation, the raw data is associated with metadata that describes its context, logs its provenance, and assures its fixity. All these forms of data need to be migrated over time from current systems to newer systems as the present systems become obsolete. Moreover, data transformation may occur within the migration process to replace obsolete formats or add new formats.





2.2 OAIS

The Open Archival Information System (OAIS) [4], an ISO standard since 2003 (ISO 14721:2003 OAIS), specifies how digital assets should be preserved for a community of users from the moment digital material is ingested into the digital storage area, through subsequent preservation strategies, to the creation of a dissemination package for the end user. OAIS is a high-level reference model and as such is flexible enough to use in a wide variety of environments; it expects more detailed steps and workflows to be developed by the implementing institution.

Of particular interest is the OAIS Information Model. This provides a high-level description of the information objects managed by the archive. An Archival Information Package (AIP) is the information package that is stored and preserved by the OAIS. It consists of the preserved information, called the content information, accompanied by a complete set of metadata, as depicted in Figure 1.

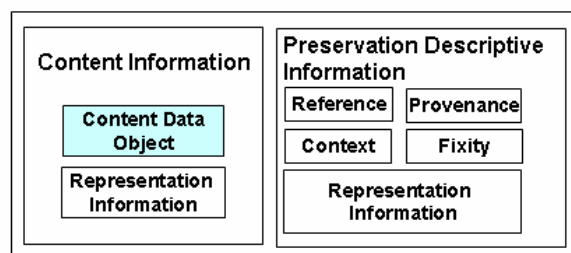


Figure 1: OAIS AIP logical structure

Note that the accompanied 'metadata' is well compartmentalized. It consists of the representation information that is required to render and interpret the object intelligible to its designated community. This might include information regarding the hardware and software environment needed to view the content data object or a specification of the data format. The other metadata, called the Preservation Description Information (PDI) is broken down by OAIS into four well-defined sections:

1. Reference information -a unique and persistent identifier of the content information both within and outside the OAIS (e.g., UUID).
2. Provenance information -the history and origin of the archived object.
3. Context information -the relationship to other objects (e.g., the hierarchical structure of a digital archive). For example, this may be a set of PDF documents, each representing a chapter of a book; or a collection of objects representing digitized images of an art collection.
4. Fixity information -a demonstration of authenticity, such as checksums and cryptographic hashes, digital signatures and watermarks.

Note that the RepInfo is a recursive object and may have additional RepInfo to interpret it. This recursion ends when facing a RepInfo that is non-digital and will be preserved by the designated community. For example, a text document represented in an Open Document Format (ODF) file is associated with RepInfo that includes the ODF specification. Assuming the ODF specification is in XML, its RepInfo includes the XML specification, and the latter is associated with a RepInfo that includes the Unicode specification. We assume that the Unicode specification is preserved by the designated community and thus it doesn't need more RepInfo.

Data Migration is another aspect of the OAIS functional model that has bearing on the storage system. Migration is the act of moving data from one system to another due to a change. This may be triggered by a variety of reasons, for example the decay of the storage media, obsolescence of hardware and software, or a change in the copyright or external environment (e.g., organization). The OAIS reference model identifies four primary digital migration types:

1. Refreshment -bit to bit copy of the entire media onto newer media of the same type.
2. Replication -copying data onto newer media which is not necessarily of the same type.





3. Repackaging -copying data while changing the placement of the components within a data object.
4. Transformation -copying data while performing format change on the data.

2.3 PRESERVATION DATASTORES (PDS) WITHIN CASPAR

The Preservation DataStores is being designed as an infrastructure component for CASPAR. CASPAR is based on OAIS and has a stack of software components, including components to: generate and access AIPs, components to capture, manage and re-construct designated community knowledge, components to extract, index and maintain the metadata, components for directory services, and the archival storage component, which includes the ultimate place of the data. The following figure illustrates a schematic view of the CASPAR system with the Preservation DataStores at the bottom, serving as the storage layer:

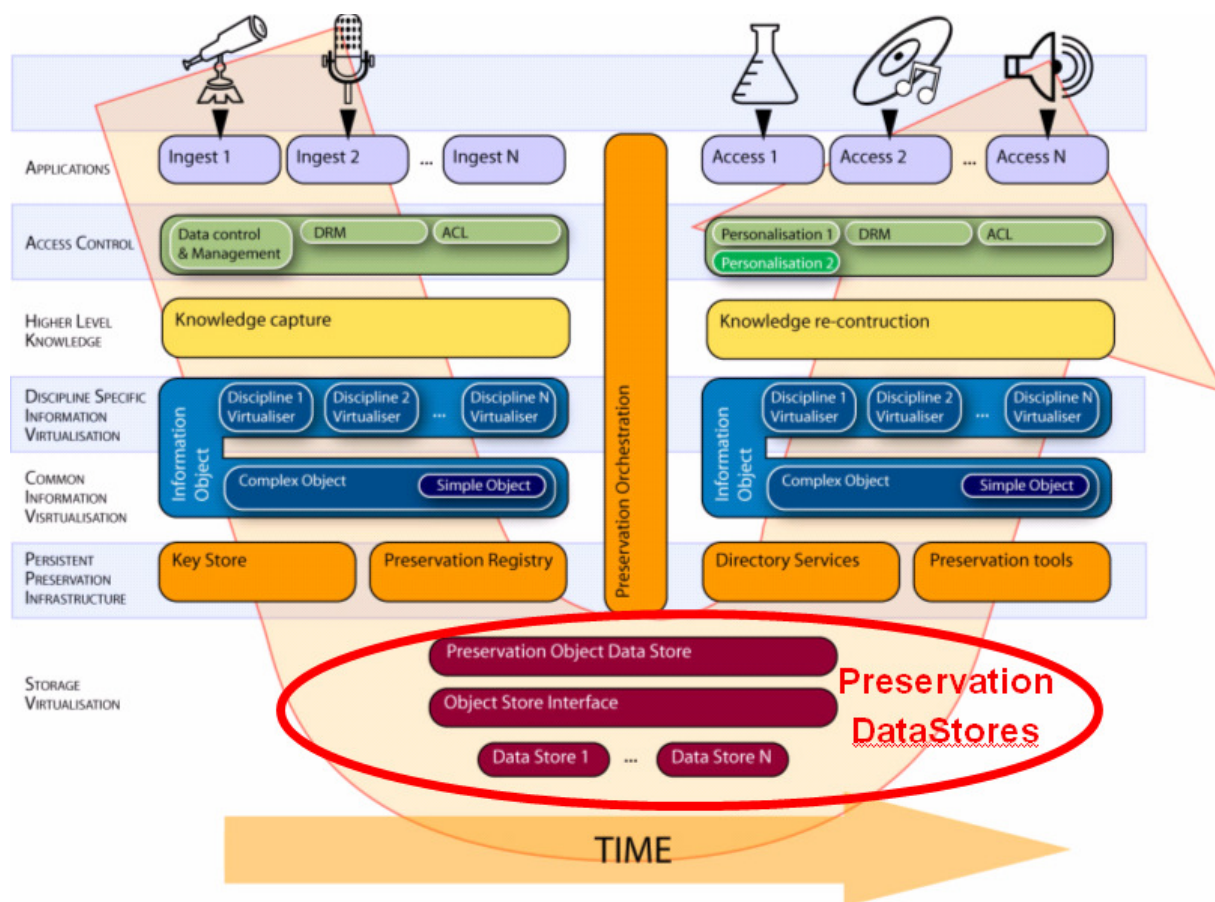


Figure 2: CASPAR System

Preservation DataStores will be utilized in CASPAR by ingesting and accessing AIPs. During ingest time, the upper layers of CASPAR generate AIPs and then provide it to the Preservation DataStores. For each AIP, the Preservation DataStores assigns or validates a persistent globally unique ID, calculates its fixity and logs the provenance. Then, the AIP is assigned to a cluster that includes AIPs intended for co-location on the same medium. The cluster assignment is performed according to some configured parameters such as clustering by the content information object format and creation date. Each cluster will eventually become a self-contained self-describing object on the media. Finally,





metadata that is needed to search for and manage the ingested AIP is stored in the Preservation Registry and Directory Service which are outside the Preservation DataStores.

During access time, the upper layers of CASPAR and search aids can be used to determine the required AIP. This AIP is then fetched from the Preservation DataStores. The upper layers then unpack the AIP and assist the user in interpreting it.

Preservation DataStores can also dynamically upload transformations modules. It then can automatically transform AIPs to new AIP versions e.g. when a format becomes obsolete.





3 PRESERVATION AWARE STORAGE

At the heart of any solution to the preservation problem, resides a storage component, which is the permanent location of the information. Traditional archival storage considers only bit preservation, if it considers preservation issues at all, and generally has functions to insert data into permanent storage, manage a storage hierarchy, refresh media on which archive holdings are stored, perform routine and special error checking, provide disaster recovery capabilities, and retrieve data from the permanent storage.

A storage system is deemed preservation-aware if it supports preservation applications. OAIS-based is a specific type of preservation-aware storage, which is based on OAIS notions, functions and information model. We argue that in order to better preserve data and understandability for long periods, a new type of storage must emerge that will take preservation considerations into account. We call this new type of storage *Preservation Aware Storage*.

Preservation aware storage should support logical preservation in addition to the traditional bit preservation. The storage should encapsulate the raw data with its complex interrelated metadata objects, so they are inseparable during the migration processes and when accessing the data in the future. The system should decrease the data transfer between the applications and the storage by offloading data intensive functions such as fixity to the storage. The system should also simplify the applications by transferring the responsibility of managing the storage-related events, such as provenance events, to the storage itself. The storage should handle migration including the ability to execute externally specified transformations.

3.1 MOTIVATION

Today, more and more storage systems offload advanced functionality and structure-awareness to the storage layer. Functions that were traditionally carried out by the application or the operating system are gradually becoming integral parts of an 'intelligent storage system'. Object-store devices (OSDs)[5], for example, offload space allocation and security to the storage device. Functions such as bit-to-bit data migration, block-level data integrity, and even encryption, are carried out by advanced, intelligent disks and tapes. Some systems (e.g. provenance-aware storage system (PASS) [6]) already track the provenance of data at the storage level rather than storing it in a standalone database.

As more intelligent archival solutions emerge, it will become possible to incorporate some of the concepts and processes specifically defined by OAIS into the intelligent storage component, resulting in OAIS-based preservation aware storage. Moreover, digital preservation systems will be more robust and have less probability for data corruption or loss if their storage component is a preservation aware storage, namely it has built-in support for preservation. The requirements of a preservation aware storage and its benefits to preservation systems are listed next.

3.2 REQUIREMENTS

The following list includes the major requirements and desired features of an OAIS-based preservation aware storage:

- In the storage, encapsulate and physically co-locate the raw data and its complex interrelated metadata objects, such as representation information, provenance, and fixity. This ensures that the metadata needed for interpretation is not separated from the raw data and thus never lost (if the raw data survives).
- Include the representation information of metadata (e.g., representation information of fixity and provenance) so the metadata can be interpreted when accessed in the future.





- Utilize the locality property and execute data intensive functions such as fixity (i.e., data integrity) computations within the storage component. This will lessen the network bandwidth and reduce the risks of data loss.
- Handle some of the provenance events internally. The applications on top of the preservation aware storage should be free of managing events that can be handled internally in the storage. Moreover, this enables richer types of provenance events and the inclusion of events related to the migration between physical medium and the transformation of representations.
- Support the loading and execution of external transformations during the migration process and facilitate on demand triggering of these transformations.
- Support media migration, as opposed to system migration. In media migration, performing migration from one system to another can be done by physically detaching the media from one system and attaching it to the new system.
- Maintain referential integrity including updating all the links during the migration process so they remain valid in the new system. This requires an awareness of certain metadata fields that represent links, both internally to the system and externally.
- Ensure readability of the data by a different system in the future. This is done by developing and supporting global self-describing media independent formats.
- Support a graceful loss of data. Some portions of the data are likely to be lost or become corrupted over time. If some data is lost, a good preservation system must minimize the economic effect of this data loss and prevent cases where data that is still intact in the system cannot be read or interpreted.

From among these requirements, we focused on supporting logical preservation; however, as should be clear, there is often an interdependency and interaction between logical and bit preservation (e.g., the interactions between migration to new media and transformations of formats).





4 PRESERVATION DATASTORES (PDS) ARCHITECTURE

Preservation DataStores (PDS) are OAIS-based preservation aware storage that focus on supporting logical preservation. They are aware of the structure of an archival information package (AIP) and offload OAIS derived generic functions such as representation information inclusion, provenance tracking, fixity computation, and migration support to the storage layer. They provide strong encapsulation of large quantities of metadata with the data at the storage level and enable easy migration of the preserved data across storage devices.

4.1 ARCHITECTURE OVERVIEW

The PDS architecture includes a stack of three layers based on OAIS, XAM and OSD, respectively. Figure 3 depicts the general architecture of PDS. The top layer is an OAIS-based preservation engine, which provides preservation functionality for heterogeneous data and applications. It includes efficient generation and placement of AIPs along with support for migration and data transformations performed within the storage. The second layer includes an eXtensible Access Method (XAM) [7] library. XAM is an emerging storage standard intended for reference information that provides a logical abstraction for data containers. The bottom layer includes an Object-based Storage Device (OSD) [8, 9], an advanced type of storage implementing an object-based interface that has a built-in access control mechanism.

In OAIS, AIPs are the information objects that are passed to and from the storage component. Thus, in PDS, AIPs are the main objects in the interface. While PDS has a generic interface, our current implementation supports both a direct API as well as a web services API. We chose a standards-based web services API since web services are platform independent and support clients built within different environments; this is an important feature in preservation environments.

Our proposed PDS is constructed of two processes. The higher level process includes the upper two layers and is responsible for implementing OAIS functions, providing the interface to the outside world, and incorporating XAM functionality. The lower level process includes the bottom layer and provides the storage in the form of the object store.

The higher level process is composed of a stack of the following components:

- Preservation Engine – provides the external interface and the OAIS specific preservation semantics.
- XAM Library – generates logical container objects of data and metadata under a common globally unique name.
- XAM to OSD – a bridge that maps the logical XAM objects to physical OSD.

Since the higher level process needs to support HTTP, web services, and security, to interface with clients of the PDS, the box may utilize an application server. Offloading an application server to the storage box supports the new paradigm of moving the application to the data instead of the traditional paradigm of moving the data to the application.



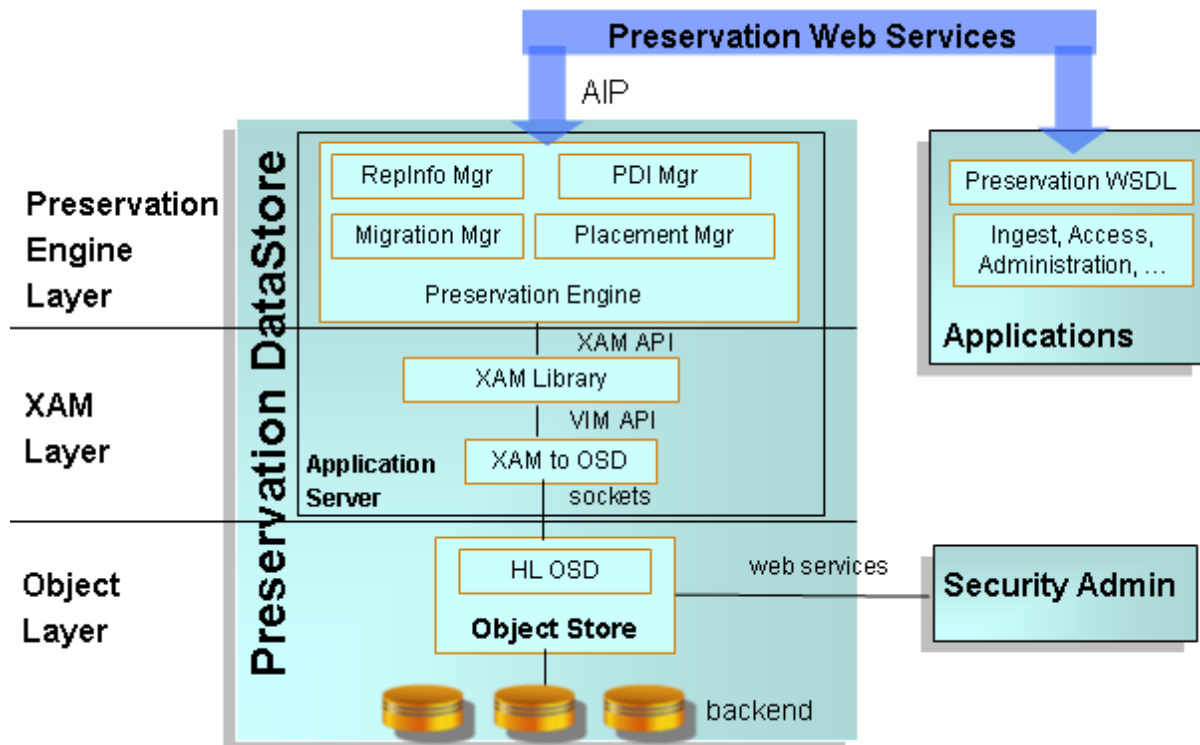


Figure 3: Preservation DataStore architecture

The second process in the PDS box serves as the object layer and includes an OSD component. The OSD process requires periodical communication with the security admin, an external component used to obtain security credentials that enforce access control. In the proposed architecture, the OSD calls a third-party security admin via an API or web services.

The object-layer can also be materialized using a standard file system as the underlying storage instead of an OSD. The chosen architecture builds on an OSD as its object-layer since an OSD is naturally designed to support an object store layer. It handles and manages the space allocation of an object and it associates attributes at the storage level in an optimal manner. This allows optimizations such as placing some key OAIS attributes (e.g., a link to the representation information) close to the data in a persistent manner. Furthermore, in cases where the actual disks are network attached, an OSD provides object-level secure access control to networked disks.

4.2 PRESERVATION ENGINE LAYER

The preservation engine component provides the external API, implements the OAIS abstractions and provides the preservation characteristics. In this section, we describe the unique features supported by the preservation engine layer and elaborate on the value of realizing them in the storage. A policy manager, either internal or external will drive the application of these features.

In addition to these specific features, a major role of the preservation engine layer is to perform a mapping between the different layers of abstraction. At the top, the preservation engine layer uses OAIS concepts to communicate with its clients. In turn, the preservation engine builds upon XAM underneath to implement its function. Thus it must map between the OAIS and XAM levels of abstraction. We describe this mapping in detail in Section 6.





4.2.1 Managing availability/data loss

One key responsibility of the preservation aware storage is to manage the availability of the data entrusted to the storage. Traditional storage systems manage data availability (e.g., RAID, remote mirroring etc.) at the block level. Instead, PDS must manage availability at a higher level, specifically at the level of an AIP. A basic assumption is that some data will be lost over time; thus a basic requirement is that the degree of information loss should be proportional to the number of bits lost. This is difficult to achieve with systems that provide availability at the block level.

Not all entities stored by the PDS are of equal importance. For example, some content data objects may have very high inherent value and some metadata (e.g., representation information for a common format) may be the key to interpreting large amounts of data. We expect that a policy manager will direct the PDS regarding the inherent importance of the content data objects while the PDS itself can determine the importance of the metadata it manages. Propagating this knowledge to the PDS allows the storage system to employ techniques such as grouping related objects and creating copies of objects, both within a medium and across media.

For composite objects such as AIPs, if any sub-component is lost, access to the entire composite may be lost. This is of particular importance when dealing with data stored using offline media (e.g., tapes), which can be physically moved between locations. To reduce the risk of data loss, the AIP sub-objects are placed together physically. The preservation engine layer aggregates the AIPs into clusters, such that each cluster is self-contained, namely AIPs that reference each other are put on the same cluster. Then, each cluster is placed on the same media unit (e.g., tape volume) to maintain physical co-location. The media unit can sometimes be a logical unit (e.g., three adjacent tape volumes), but we avoid situations where the data is completely distributed. In addition to this co-location of related data, the PDS may also make physical copies of information that is deemed of high importance.

The alternative to having the PDS manage the availability of AIPs would be to have the client of the storage manage the physical organization of the data. Such a breaking of abstraction would greatly increase the complexity of the client and would be a step backwards since in most applications the application is not aware of the physical data placement.

4.2.2 AIP transformations

Another key aspect of long term digital preservation is the need to migrate data objects being preserved. This may be triggered by changes such as media decay, obsolescence of hardware or software, or a change in the copyright or external environment (e.g., organization). Some types of migration entail simply copying a set of bits from one medium to another to ensure the bits are preserved. It is natural for this type of migration to be handled by the storage, since the storage is aware of the media characteristics and can monitor the health of the media. It also saves bandwidth because the data does not need to be read by a client and written by the client to a different media.

Another type of migration is transformation, which is the focus of logical preservation. Transformations involve changing the bits of an existing data object (e.g., transforming data in a format that is about to become obsolete into the replacing format). While the storage should not be aware of the semantics of the transformation, it is natural for the storage to apply the transformations. This can be done at the same time as data is migrated for bit preservation, saving accesses to the physical medium. Alternatively, it can be done on demand; prior to returning an AIP to the client, the format can be transformed into one the client recognizes. The storage should support transformations since this is a data intensive function and it can improve overall system performance by performing the computation for the transformation close to the storage. It also minimizes the risk of data loss involved in massive data transfers for transformation purposes.

4.2.3 AIP identifier generation

Each AIP is assigned with a persistent globally unique identifier. In PDS, this identifier is constructed of the following parts: a logical ID which is the basic identifier of the AIP and can be used to find the





AIP in the preservation system, a version number that identifies the various versions originating from the same AIP, and a copy number that identifies the various copies of the same bit-wise AIPs.

It is important for the AIP ID generation to be supported by the preservation storage since at times it will create new AIPs. The newly created AIP may be a version or a copy of another AIP in the system, in such a case it will share the logical ID of its originating AIP and have a different version or copy number. In other cases, a brand new AIP may be created in the PDS, thus it will have a new logical ID.

An event that triggers the creation of a new version to the AIP may be a change to the Content Information Object (e.g., a transformation causes changes to the data and its RepInfo). External PDI events also cause the generation of a new AIP (e.g., a Provenance event that documents a change of the AIP ownership). Note that technical PDI events (e.g., a Provenance event that documents media migration) will edit the PDI data but will not cause the creation of a new version AIP.

A case in which a new logical ID needs to be generated by the PDS may be the preservation of the RepInfo of an AIP. To preserve RepInfo, a new AIP is created with a new logical ID. When ingesting an AIP with complex data, it may need to be preserved in several AIPs aggregated in a single AIC. Each of those AIPs, as well as the AIC will be assigned with a newly generated logical ID.

4.2.4 Storlet container

Digital preservation systems need to periodically perform data validation and data transformation procedures. To perform these data-intensive tasks, traditional systems move the data across the network to the application side. Once the procedures complete, the system moves the data back to the storage component. Similar to the ABACUS system [10] and active disks [11], we propose to better utilize the locality property and perform these data intensive procedures within the storage. Therefore, the preservation engine includes a module container that can embed and execute restricted modules with pre-defined interfaces. We use the term ‘storlets’ (like applets in applications and servlets in servers) to describe these deployable restricted modules. Storlets, such as transformation modules and fixity computation modules, can be deployed in PDS and later on executed either periodically or when explicitly triggered.

4.2.5 Manage preservation specific metadata

OAIS specifies a set of metadata that is needed for AIPs. By having the management, and in some cases the computation, of this metadata handled by the storage component, we can offload work from the client, avoid transferring data back and forth between clients and the PDS, and ensure a consistent and correct implementation

One important set of metadata is the Preservation Description Information (PDI). The PDI includes the fixity (an integrity check value). Computing the fixity within the PDS prevents unnecessary data transfers and ensures a consistent implementation for all objects. In addition, the PDS will be executing data transformations, which change the fixity value; hence, we need to know how to calculate fixity. The fixity algorithm may be updated by the user (using the storlet container) or chosen when there are several options.

Provenance is a second class of metadata in the PDI. Provenance events may be triggered internally (e.g., replication, transformation, etc.), making it natural to manage the provenance data within the storage. Other events, such as a change of ownership, are triggered from outside the archival storage and an API is called to inform the preservation engine of the event. If the AIP was already moved to offline media, the preservation engine records the provenance events that occur between migrations and is responsible for updating the preservation data on the next migration. If the AIP is online, the updates may take place upon occurrence.

Another important class of preservation-specific metadata is Representation Information (RepInfo), which is the metadata needed to interpret the content data that is being preserved. The RepInfo is by itself an AIP, which can be referenced by multiple other AIPs. The RepInfo includes references to additional RepInfos needed to interpret it. This forms a RepInfo network where some of its links may





refer to RepInfos stored in external registries. To ensure the availability of the data it manages, the PDS validates the external RepInfo links and optionally copies them from the external registries to the PDS to maintain physical co-location (as described above). Multiple AIPs with the same RepInfo should be clustered together on the same medium. This allows the system to share the RepInfo among content objects with the same representation and manage the availability of the RepInfo by creating the appropriate number of copies. Since the storage layer is aware of the mapping of data to physical media, it is natural to have the management of RepInfo offloaded to the storage.

The Preservation Description Information (PDI) has a special kind of RepInfo. This is the representation information for the metadata generated by the preservation engine itself and thus should be managed by the storage component.

The PDS also ensures the referential integrity of the links in the PDI and RepInfo elements. This includes links to external registries, URLs, and AIP IDs. Each of these references needs to be validated. This validation should take place on ingest, access, and each time the AIP is accepted into the system (during ingestion, transformation etc.). This validation of referential integrity is essential in ensuring the availability of the data.

4.3 XAM LAYER

While PDS exposes high level complex logical objects such as AIP, existing storage systems (object, file or block) expose much lower-level interfaces and do not provide the necessary abstractions. Our architecture uses a middle-layer of abstraction to mediate between the lower storage system and the preservation engine. We chose to base this mid-layer on Extensible Access Method (XAM).

XAM is a Storage Networking Industry Association (SNIA) initiative to define a standard interface between consumers (application and management software) and providers (storage systems). XAM Specification defines three primary objects: XSet, XSystem and XAM Library. The XSet, which is the fundamental artifact in XAM, is the basic unit of data for application to commit to persistent storage. It is a data structure that is a package of multiple pieces of data and metadata, bundled together for access under a common globally unique external name, called an XUID. The XSystem, a logical container for one or more XSet records, serves as a virtual storage to XAM applications. The XAM Library enables applications to discover and communicate with XSystems.

Data can be attached to each primary object, in a form of XAM field. Each field has a unique name in the scope of its primary object. There are two types of XAM fields:

- Properties – fields that usually include metadata and thus will be indexed and used in queries. Their type is a “simple” type and is one of Boolean, Int64, Float64, String256, DateTime, XUID.
- XStreams – fields that include unbounded byte streams. Their type is a valid MIME-type.

Each XAM field has a fixed set of attributes for its content and behavior:

- Value – The actual value (content) of the field.
- Type – The type of the value of the field; namely simple type for Properties, MIME-type for XStreams.
- Readonly - A Boolean value indicating whether the field can be modified by an application.
- Length – The actual size of the field value in bytes.
- Binding – A Boolean value indicating whether the field is immutable within the XSet. If true, then field modification causes the automatic creation of a new XSet with a different XUID. This attribute is relevant only to XSets.

The XAM architecture allows applications to use the XAM API to store and retrieve information in a vendor-independent and location-independent manner. The top software module in the XAM architecture is the XAM library, which exposes the XAM API. This module interacts with the application and provides a view of the underlying storage through the entities of the XAM world. Under the XAM library resides the Vendor Interface Module (VIM), which acts as a bridge between the XAM standard APIs and the vendor storage systems.





We chose to use XAM in PDS since it adequately addresses many of the special needs of a preservation aware storage. Choosing a standard rather than inventing an internal layer is in line with the basic design principle of presentation systems; open standards are more likely to be robust, system-independent, last longer, and support interoperability. The following preservation special aspects are addressed by XAM:

- Importance of Metadata - In preservation aware storage, the metadata is crucial for the interpretation of the data upon retrieval; therefore metadata is as important as the data. XAM enables bundling large amounts of data and metadata in an XSet and handles data and metadata in the same manner by representing both as XAM fields. XSet may serve as a smart "building block" to support the AIP complex structures.
- Migration of Data - Preserved data is expected to be migrated over the years and possibly transformed to support the obsolescence of formats, hardware or software. Each XSet is associated with a globally unique ID, which enables the stored data to be location independent. Furthermore, XAM has a built-in import/export service for XSets. This in turn allows transparent media and technology refresh cycles and facilitates data migration from one system to another over time.
- Accumulated Data - Preserved data is accumulate rather than overwritten. XAM is specialized for reference data (i.e., written once and primarily referenced later). As mentioned above, each XSet field has a binding attribute. This can be leveraged as a build-in versioning mechanism which provides an easy way to generate new versions of the content while keeping the old version of the content intact.
- Complex operations - Operations such as ingest are expected to be comprised of many sub-operations on large amounts of data and metadata. XSets are viewed a single transactional unit that can either be committed as a whole or abandoned. This transaction mechanism of XAM simplifies the implementation of operations such as ingest as single transactions. Sharing Data
- Across Time and Space - The media containing the preserved data is expected to be shared across time and space. XAM has a built-in import/export service based on a standardized self-describing format for XSets. We leverage this service to create a self-describing, self-contained format (SD-SCDF) for AIPs written to a portable medium. The self-describing property allows the AIPs to be extracted from the medium and the self-contained property ensures that the necessary information is available to interpret the data.

4.4 OBJECT LAYER

As described in the previous section, the XAM standard provides us with a high-level storage abstraction needed for the preservation engine. The XAM library interacts with the supplied underlying storage system via the VIM API. The VIM maps XAM entities such as XSystems, XSets, XSet fields and their attributes to the vendor storage system entities.

The selection of the storage system is driven by its compliance with XAM and, in addition, must also support the special needs of a preservation aware storage. While we intend to architecturally support both file systems and object stores via the cross-VIM portability of XAM, we believe that Object Stores (OSD) are better suited to the special needs of preservation aware storage:

- In an OSD, metadata is an integral part of the object and is managed, stored persistently, and recovered with the object's data. The fact that the importance of the metadata penetrates all the way to the object layer is an advantage for the preservation aware storage.
- The OSD security model provides security per object. This enables the use of networked storage while supporting fine-grained secure access control to the storage. In a preservation system, which stores vast amounts of data, the ability to store it on a scalable storage network is essential.
- OSD enables us to include application hints that will denote OAIS "hot" attributes such as the attribute that links to a RepInfo object. The OSD will use these hints to improve storage performance and robustness.





- OSD is an industry standard. Standards are preferred in preservation aware storage in order to facilitate interoperability.
- OSD provides a natural and adequate mapping to XAM.





5 INTEGRATING PDS WITH SRB/IRODS AND EXISTING ARCHIVES

This section describes the integration of PDS with existing prominent archive technologies such as SRB/iRODS as well as suggests how to support data that already resides in existing file systems and archives but need to be preserved for decades and more.

5.1 PDS AND SRB/IRODS

The Storage Resource Broker (SRB)/ Intelligent Rule-Oriented Data management System (iRODS) [12, 13, 14, 15] is a data grid technology developed and owned by the San Diego Supercomputing Center (SDSC). It manages distributed data, enabling the creation of data grids that focus on the sharing of data, and was recently extended to *persistent* archives that focus on the preservation of data. Data grid technology provides the fundamental management mechanisms for distributed data in a scalable manner. This includes support for managing data on remote storage systems, a uniform name space for referencing the data, a catalog for managing information about the data, and mechanisms for interfacing to the preferred access method. The SRB/iRODS is middleware software - it builds on top of standard file systems, commercial archives and storage systems.

Over the past years, SRB has been used as a foundation technology for providing a persistent archive in the context of preservation projects. Many of these projects were commissioned by NARA, the National Archives and Records Administration, and supported by the Library of Congress and NSF. As a persistent archive, it managed the retention of the digital record as well as the context that describes the origin, relevance and authenticity of the record.

SRB handles media failures, data mirroring and distribution of data. It stores the data records as files on the storage system and assumes that the data object is packaged into an AIP. The AIP is written to the storage repository but a separate database is used to store the metadata related to the electronic record.

The basic architecture of SRB consists of an SRB server and a Metadata Catalog (MCAT) server. The SRB server exposes various file-like APIs to the application and interacts with the storage system. The MCAT server handles the information stored in the SRB database. The picture below is taken from SRB documents. The iRODS adds to it a rule-based system, which allows the user to make assertions about the rules under which the data is being maintained.

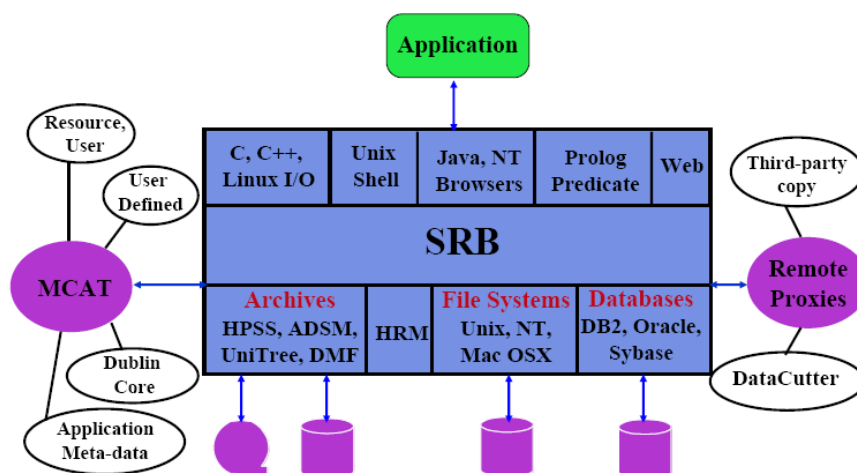


Figure 4: SRB system





PDS layers can be integrated with SRB/iRODS both on top and in the bottom (see figure 5).

Top layer - will extend today's SRB/iRODS APIs which are very similar to file operations, to support PDS API for preservation applications. This will utilize PDS Preservation Engine layer.

Bottom layer - to support storage constructs that are suitable to store objects (OSD, XAM devices). This will utilize PDS XAM and Object layers.

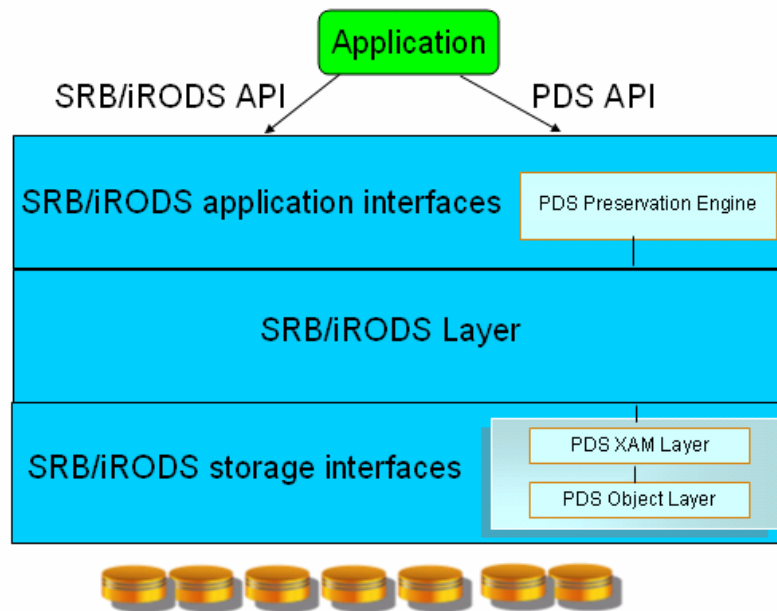


Figure 5: PDS and SRB/iRODS

5.2 PDS AND EXISTING ARCHIVES

In many cases, the data that is subject to long term digital preservation already resides in existing archives. These archives may be simple file systems or more advanced archives that include enhanced functions such as metadata advanced query, hierarchical storage management, routine or special error checking, disaster recovery capabilities, and bit preservation. Some of this data is generated by applications that are unaware of the OAIS specification and the AIP logical structure, and generally include just the raw content data with minimal metadata. While these archives are appropriate for short term data retention, they cannot ensure long term data interpretation at some arbitrary point in the future; at this point everything can become obsolete including hardware, software, processes, format, people, and so forth.

PDS can be integrated with existing file systems and archives to enhance such systems with support for OAIS-based long term digital preservation. Figure 6 below depicts the architecture for such integration. We propose the addition of two components to the existing archive: an *AIP Generator* of manifest files and a *PDS box*. The *AIP Generator* generates AIPs, where each AIP is a manifest file with links to the existing content data and other metadata that already exists in the archive. If some metadata is missing (e.g., RepInfo) the AIP Generator will be programmed to add that part either by embedding it into the manifest file or as a separate file or database entry linked from the manifest file. Sometimes, programming the AIP Generator to generate those manifest files can be quite simple, for example, if there is an existing naming scheme that relates the various AIP parts.

The AIP Generator can be triggered to generate a manifest file either by the existing archive or by the existing Data Generation Application. The former method is possible if the existing archive has the mechanism to trigger an event when a new content data is ingested into the archive. The latter method





is possible if the existing data generation application has the mechanism to hook the AIP Generator when a new content data is generated. Note that in both cases, data can be entered into the archive using the existing data generation applications and will thus not require writing new applications.

The generated manifest AIPs are ingested into the second component – the *PDS box*. PDS provides most of its functionality including awareness of the AIP structure, execution of data-intensive functions such as transformations within the storage, handling technical provenance records internally, support for media migration, and maintenance of referential integrity. However, PDS cannot of course guarantee the intelligent data placement. It cannot guarantee physical co-location of the various parts that constitute the AIP and, likewise, it cannot guarantee the aggregation of related AIPs into clusters placed on the same media unit. However, when the next periodic migration phase happens because of media decay or for other reasons, PDS can optimize the data placement and provide physical co-location of related AIPs.

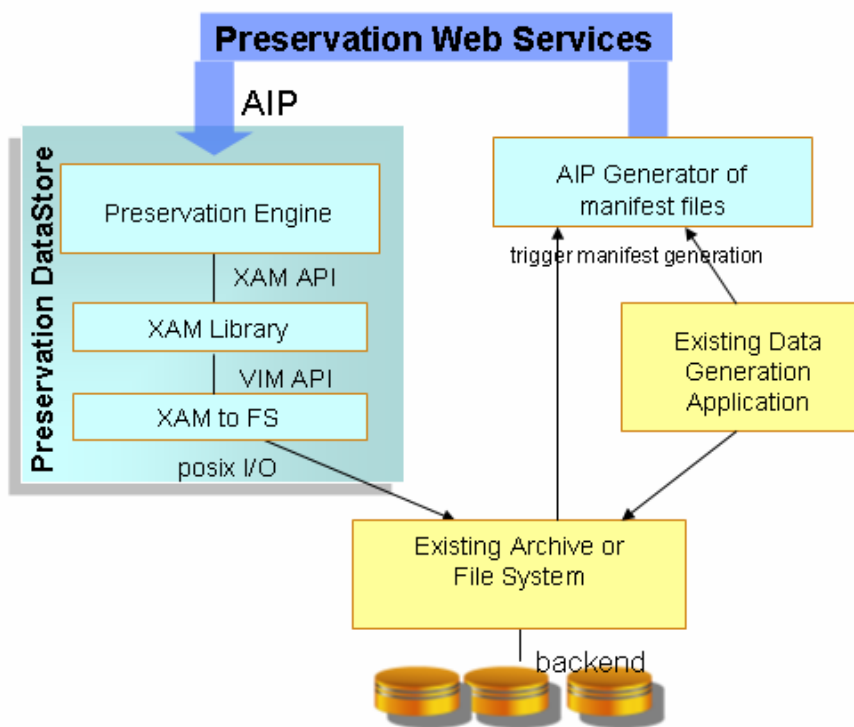


Figure 6: Integrating PDS with existing archives or file systems





6 AIP REPRESENTATION IN PDS AND MAPPING

A key point in the PDS architecture is the use of standards in each of the layers. In the Preservation Engine layer, we exploit the OAIS reference model as the top level standard. We implement OAIS concepts utilizing the XAM emerging standard, which in turn is mapped to the OSD standard at the object layer. As a result, the complex structures of the OAIS implemented in the high-level Preservation Engine layer are gradually mapped to the simpler structures of the object layer. In this section, we describe the representation of OAIS objects in the Preservation Engine layer and propose their mapping to the underlying layers. More specifically, we describe the representation and mapping of an AIP.

6.1 AIP REPRESENTATION IN PRESERVATION ENGINE

We define a hierarchical representation of an AIP. The basic building-block of an AIP is the Information Object, which consists of a single instance of content data, the raw data being preserved, and RepInfo to interpret the data. In the PDS architecture, the RepInfo is also represented by an AIP. This is done for two main reasons. First, being a shared resource in the archival storage, the RepInfo element has to be an independent unit identified by a unique ID. Second, RepInfo objects consist of content data and RepInfo (forming the recursive RepInfo network), and also need to be preserved. By representing RepInfo with an AIP, we can utilize an existing mechanism to preserve the RepInfo, while keeping the design simple. Consequently, each Information Object contains embedded data for its content and an AIP reference for its RepInfo.

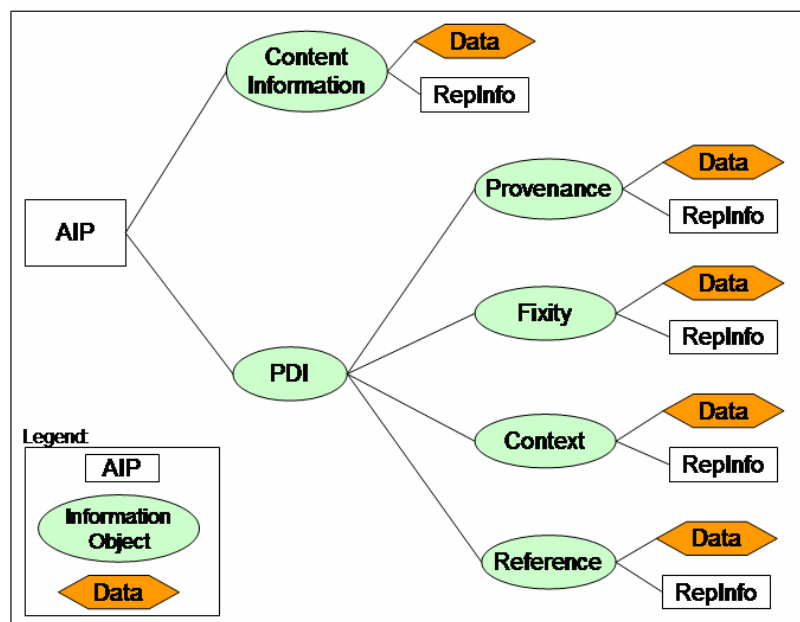


Figure 7: Representation of AIP in Preservation Engine layer

As depicted in Figure 7, an AIP contains elements of two types: Content Information and PDI both inherit from the Information Object. Content Information is a simple Information Object. A PDI object is a more complex Information Object, containing four sub-elements (Provenance, Fixity, Reference and Context). OAIS does not specify whether these sub-elements are themselves Information Objects. In the PDS system, we decided that the PDI sub-elements should inherit from Information Objects because they contain data and RepInfo to interpret that data. For example, provenance data is kept in a certain format; the description of this format serves as the RepInfo of the provenance data. In other





words, by representing the elements of the PDI themselves as Information Objects, we enable the PDS to use the same mechanism to preserve user data and its own metadata.

6.2 PRESERVATION ENGINE MAPPING TO XAM

From a high-level perspective, an AIP is naturally mapped to an XSet. The AIP contains pieces of data and metadata bundled together under a unique ID. Recall that the XSet offers a unique ID (XUID) and may contain pieces of data and metadata stored as its XSet fields.

Looking at the AIP content, an AIP is constructed of Information Objects. Within an Information Object, the content data (an opaque byte stream) is naturally mapped to an XStream. The RepInfo reference is mapped to a property of type XSet reference (XUID), since each RepInfo is represented by a separate AIP and therefore mapped to a separate XSet.

The challenge lies in mapping the Information Object itself. How do we tie together its contained objects to enable a sound mapping of the AIP inner-structure while using XAM correctly? The AIP has a hierarchical and complex inner-structure, while XAM is oriented towards a flat structure and aims to contain many XSet fields in a single XSet.

One option is to map each Information Object to an XSet. In this option, the AIP is mapped to an XSet that may contain two XSet references – one for the Content Information and one for the PDI. The PDI XSet, in turn, contains four XSet references (Provenance, Fixity, Reference and Context). Although true to the AIP structure, this mapping enforces the creation of many sparse XSets and thus does not follow the XAM spirit. Namely, it does not exploit the advanced encapsulation capability of XAM while still paying the overhead of maintaining complex abstractions.

We chose to implement an alternative option in the PDS. We avoid mapping the Information Objects and place their contained objects directly under the AIP XSet as XSet fields. In this manner, the AIP hierarchical structure is completely flattened. A naming scheme maintains the grouping of XSet fields to Information Objects (see Figure 8). This mapping better exploits the XAM object structure (taking advantage of the fact that an XSet is a container of multiple fields of multiple types). Moreover, mapping an AIP to a single XSet enables transactional operations on AIPs. Operations on AIPs (e.g., ingest AIP) are complex and composed of many sub-operations. Having the AIP as a single transactional unit guarantees executing each complex operation on an AIP as a single transaction.

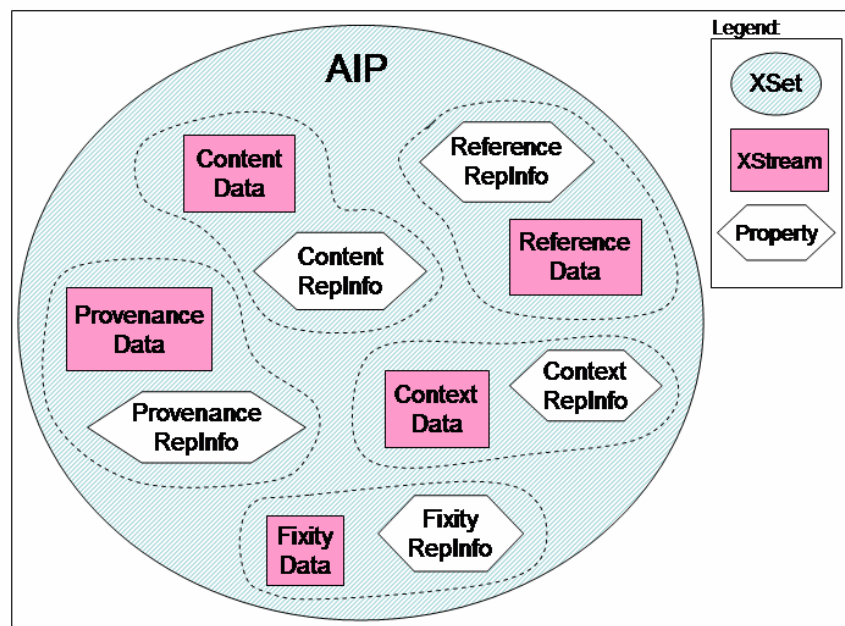


Figure 8: Flat mapping of AIP to XSet





Above, we suggested that each content data naturally maps to an XStream, assuming that the content data is an opaque byte stream. This assumption is correct for the Content Information data. However, the PDI sub-objects' data have a complex inner-structure. Fixity and provenance data consist of a series of events, whereas context data contains different pieces of data and references. If each such data is mapped to a single XStream, we will incur additional complexity in parsing these XStreams on each access. Nevertheless, such an implementation would keep the mapping simple.

An alternative is to map each such data to a group of XSet fields, parsing the data at the XAM level. This means each piece of data or reference (e.g., each provenance event) is mapped to a separate XSet field. The mapping is more complicated, but enables easy search and access to these pieces of data and reference.

We also looked into which XAM attribute values should be assigned to each XSet field, in particular the use of the "binding" vs. "non-binding" qualifier of XSet fields. Recall that by setting a binding attribute to a field, XAM implies that any edit/addition/removal of this field results in the automatic creation of a new XSet. Hence, the binding qualifier determines the conditions and/or events under which a new AIP will be generated.

Since any change to the Content Information will result in the creation of a new version of the AIP while keeping the original AIP intact, we map the Content Information's content data and RepInfo as binding objects. The RepInfo of the PDI sub-objects are also set to binding. For example, a change to the fixity RepInfo may be an update of the fixity algorithm and should cause the creation of a new version of the AIP.

On the other hand, the handling of PDI sub-objects' data is more complex. For example, we believe that unlike external provenance events (e.g., a change of ownership) that should cause the creation of a new AIP, the internal generation of provenance events should not. Such an internal event may occur during media migration, where a provenance event is added by the Preservation Engine to document the migration. Therefore, if provenance data is mapped to a single XStream, it should map as non-binding. When external events are added, the creation of the new XSet must be handled by the Preservation Engine layer since it will not be automatically handled by XAM. If each provenance event is mapped to a separate XStream, external events map as binding while internal events map as non-binding.

One benefit of the use of XAM as described above is in supporting a self-describing, self-contained data format (SD-SCDF) for AIPs written to a portable medium. A cluster of AIPs is mapped to a XAM object (XSet) with properties that include references to the various AIPs in the cluster. By that, we take advantage of the XAM export format and propose that one way to implement SD-SCDF will be to use XML-binary Optimized Packaging (XOP). The SD-SCDF will include the references and offsets to the various AIPs of the cluster, followed by an attachment that includes a serialization of each AIP. This allows parallel access to the AIPs if desired. The serialized AIP will follow the mapping to XAM objects and include a description of the compact parts of the AIP (e.g. references to the various RepInfos), followed by a table of contents attachment to list the offsets to each of the streams. The table of contents is followed by a MIME attachment for each one of the streams (e.g., content data object stream, reference data stream, provenance stream, context stream, fixity stream).

Note that the SD-SCDF format has by its own RepInfo that needs to be preserved, such as the specifications of XOP, MIME, XAM, etc.

6.3 XAM MAPPING TO OSD

There is an ongoing effort in the OSD working group in SNIA to propose a mapping of the XAM standard to OSD. To date, the work that has been done offers a mapping of XAM objects and their attributes to OSD objects and attributes. Other issues, such as security, have not yet been addressed.

The XAM to OSD mapping to be used in the PDS system is based on the SNIA proposal and is generic; namely, it has no preservation specific characteristics. It aims to provide a natural and efficient backend support to XAM.





7 PDS INTERFACES

The PDS interfaces expose the PDS entry points and also a set of interfaces to support these entry points. The entry points may be called directly or via web services. The PDS interfaces aim to be abstract, technology independent and to survive implementation replacements.

7.1 INTERFACE OVERVIEW

The entry points are defined in the set of PDS manager interfaces. The set of PDS manager interfaces includes the following: PDSManager, PDSIntegratedManager, PDSMigrationManager, PDSPdiManager, PDSRepInfoManager and PDSPackagingManager.

Other interfaces are used by the entry points as arguments and/or return values. The other interfaces are basically data structures used by the PDS entry points. Their methods are not exposed as web services and are called locally to build the arguments required for the different entry points and to retrieve the values returned by API calls. When using the PDS entry points via web services, these structures are streamed into XMLs.

The API may return different exceptions that are also PDS interfaces.

Some of the structures used by the PDS entry points, such as the PDI sub-section records' inner-structure (e.g. the inner structure of a Provenance record) may have different variants, depend on the source that generated the record. PDS aims to treat a set of records that may differ in their inner-structure in a harmonic way (e.g. the set of Provenance records may contain records with different inner-structure; still PDS should treat them as a set of equivalent records). To enable that, the inner-structure of each record is defined in an XML schema (i.e. structural RepInfo for each record) kept along with the record's content. When a record is generated by PDS we use the default schema for the record. These default schemas will also be exposed to the user if wishes to use them.

The following table is a summary of PDS interfaces.

Table 2: PDS manager interfaces summary

Interface Summary	
<u>PDSIntegratedManager</u>	PDSIntegratedManager interface defines the entry points that should be implemented in an environment in which PDS is integrated into an existing system.
<u>PDSManager</u>	PDSManager interface defines the entry points implemented and exposed by PDS.
<u>PDSMigrationManager</u>	PDSMigrationManager interface defines the entry points that relate to different migrations of AIPs.
<u>PDSPackagingManager</u>	PDSPackagingManager interface defines entry points that enable extending and querying the PDS packaging capabilities.
<u>PDSPdiManager</u>	PDSPdiManager interface defines the entry points to access and manipulate the PDI sections of AIPs that were already ingested into PDS.
<u>PDSRepInfoManager</u>	The PDSRepInfoManager interface defines the public methods for the Representation Information Manager.

The following diagram describes the inter relationships within the PDS interfaces:



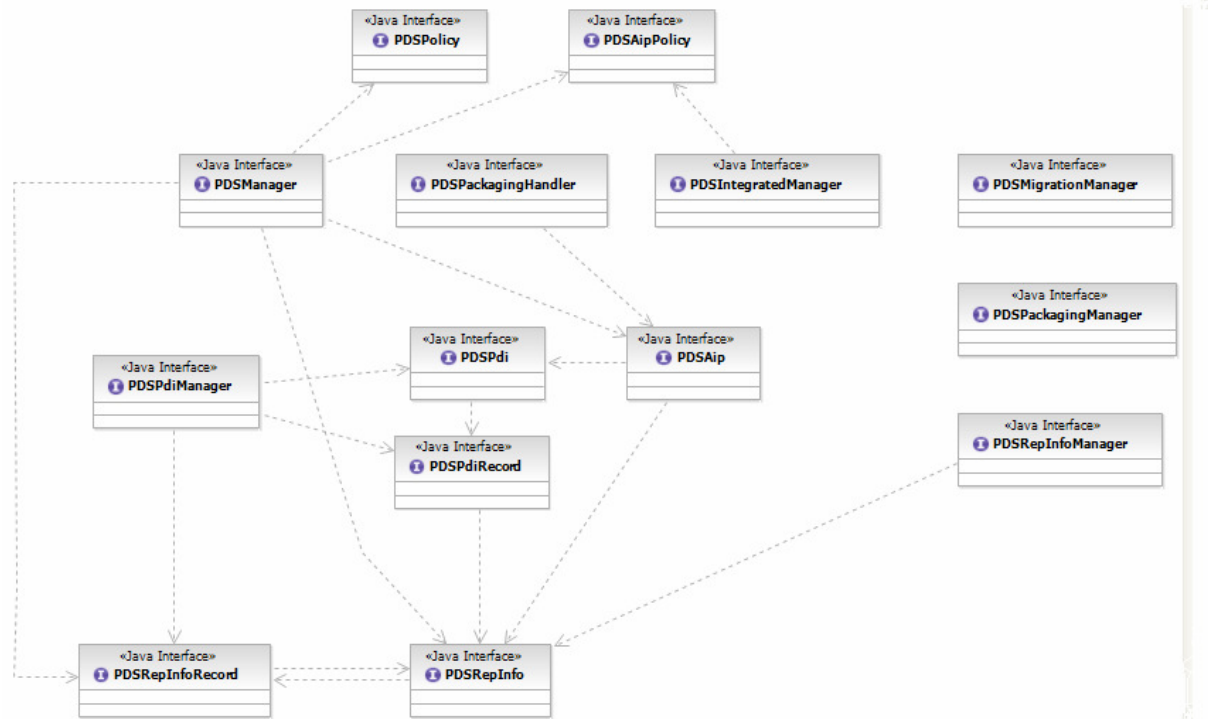


Figure 9: PDS interfaces

7.2 PDSMANAGER INTERFACE

PDSManager interface contains the main entry points that PDS users may call in order to preserve AIPs. The entry points' parameters and return values are interfaces as well and provide methods to manipulate their fields.

These interfaces deal with preservation of AIPs and handling their content data and content data RepInfo.

Passing arguments:

Some of the entry points use PDSLink to reference their arguments and/or their return values. This enables to put the data at a staging area and pass a PDSLink that contains a link to download the data from. Other entry points send and receive the data directly as a byte stream.

Groups of entry points:

PDSManager entry points are divided into groups, each group is responsible for a different PDS function and may includes several variants on that function. The groups are the following:

- ingest AIP - implements different ways to ingest an AIP or an AIC to PDS
- access AIP - implements access an AIP or to its content data and content data RepInfo sections
- handle AIP - enables manipulating AIPs that were ingested priority
- query AIP - performs queries on the AIPs in the system. Note that currently we support only one query; in the future different queries may be defined by passing different arguments to the query entry point described below





- load validation - loads content validation modules into PDS to be executed later
- handle policies - enables manipulating PDS policies
- informative entry points - provide information about the PDS system

Packaged and unpacked AIPs:

The AIPs may be ingested and accessed either in a packaged manner, or in an unpackaged manner. A packaged AIP will be unpacked on ingest and packed on access by PDS according to the packaging format argument passed by the user. When using an unpackaged AIP, PDS interfaces should be used to build the different required arguments.

The ingest functions enable the ingestion of both AIUs and AICs. A packaged AIP will be unpacked within the PDS and handled in the same way whether it is an AIU or an AIC. In case of an unpackaged AIP that is actually an AIC, first all AIUs contained in the AIC should be ingested separately, and only then the AIC should be ingested, containing the set of AIP IDs as its content.

Preservation policies:

The PDSAipPolicy interface allows changing the preservation settings for an ingested AIP by passing policy arguments upon ingest. In case no such arguments are passed, the PDS will use the default policies for the preservation of the AIP.

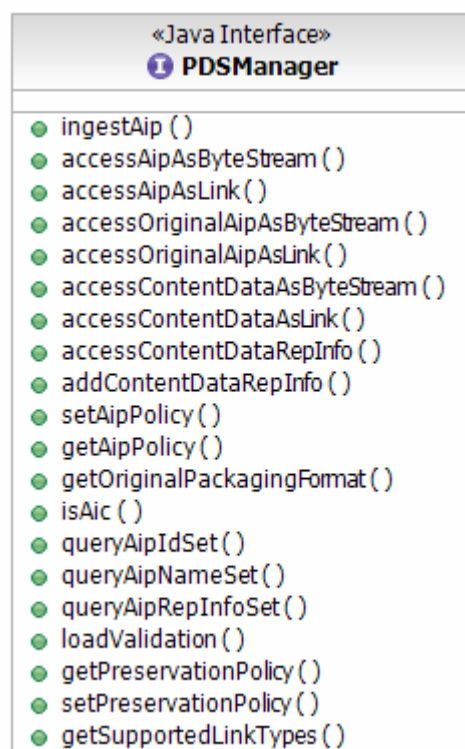


Figure 10: PDSManager interface

AIP IDs:

Each AIP is assigned with a globally unique identifier called "AIP ID". The AIP ID can be either generated by the PDS or externally. If the AIP ID is generated externally, it should be passed to the ingest entry point through an implementation of PDSAipId interface; PDS will then validate this ID internally. If a PDSAipId argument is not passed on ingest, PDS will generate the AIP ID internally. On methods that request an AIP ID as a parameter, the AIP ID





provided may contain only a logical AIP ID. The default version number is the newest version of the AIP. If the function edits the AIP, the edit will be performed on all copies of the AIP.

The UML diagram in figure 10 summarizes the PDS entry points that are defined in the PDSManager interface:

7.3 PDSINTEGRATEDMANAGER INTERFACE

PDSIntegratedManager interface contains the entry points that PDS users may call to ingest an AIP for a document that is already stored in the existing system. See figure 11 for the UML diagram of this interface.



Figure 11: PDSIntegratedManager interface

7.4 PDSMIGRATIONMANAGER INTERFACE

PDSMigrationManager interface contains the entry points that PDS users may call to perform migrations and transformations on AIPs and to load new transformation modules into PDS to be executed later on. See figure 12 for the UML diagram of this interface.

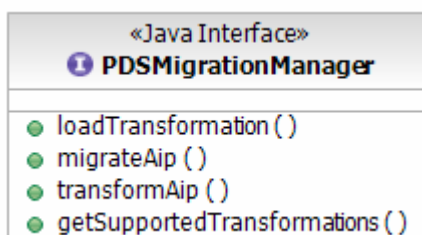


Figure 12: PDSMigrationManager interface

7.5 PDSPDIMANAGER INTERFACE

PDSPdiManager interface contains the entry points that PDS users may call to access each PDI sub-section and its RepInfo, and to add additional records and RepInfo to each. In addition a user may load a new module that contains an implementation of a fixity algorithm not yet supported by PDS and to extend the fixity capabilities of PDS this way. See figure 13 for the UML diagram of this interface.

7.6 PDSREPINFOMANAGER INTERFACE

The PDSRepInfoManager interface allows initializing the RepInfo system from a file and find/add new RepInfo items. See figure 14 for the UML diagram of this interface.



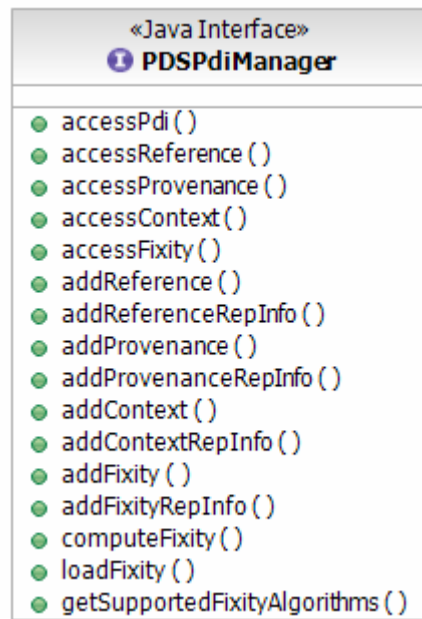


Figure 13: PDSPdiManager interface



Figure 14: PDSRepInfoManager interface

7.7 PDSPACKAGINGMANAGER INTERFACE

PDSPackagingManager interface contains the entry points that PDS users may call to load a new packaging module and to query the existing packaging modules in PDS. A new packaging module must implement the PDS PackagingHandler and PackagingHandlerCreator interfaces.

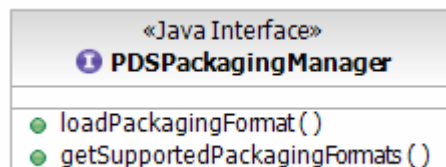


Figure 15: PDSPackagingManager interface

7.8 PDSAIP INTERFACE

PDSAip interface represents an AIP. This interface enables building an unpackaged AIP. An unpackaged AIP is a container of all the different AIP sections in a structure that is understood by the PDS, enabling PDS to perform actions on different sections of the AIP.

This interface provides methods to add Content data either in the form of byte stream or in the form of





a link.

Additionally, it enables adding RepInfo to interpret the Content data and PDI sections.

The following diagram in figure 16 demonstrates the structure of the PDSAip interface that is used for working with AIPs that are not packed in a special packaging when submitted to PDS:

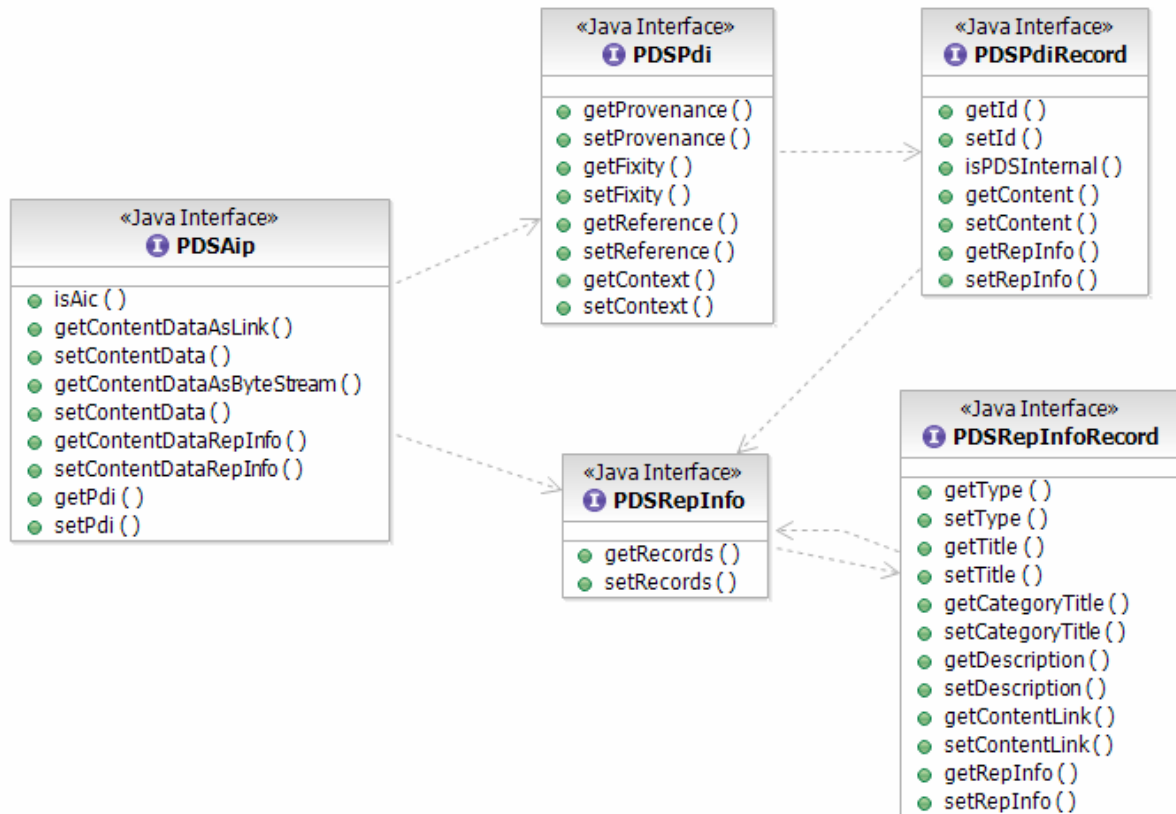


Figure 16: PDSAip interface and its related interfaces

7.9 PDSPDI INTERFACE

PDSPdi interface enables access to the different PDI sub-components. It is used to make an abstraction to the PDI inner-structure to prevent the need to change APIs in case the PDI inner-structure changes. See figure 16 for the UML diagram of this interface.

7.10 PDSPDI RECORD INTERFACE

PDSPdiRecord interface represents PDI records (PDSProvenanceRecord, PDSFixityRecord, PDSReferenceRecord, PDSContextRecord).

The following attributes are assigned to each record:

- A unique ID within the scope of its AIP section record set (Provenance, Fixity, Reference or Context)
- Indication whether the record was generated internally in the PDS or externally
- RepInfo to interpret the structure and/or semantics of the record





Each record has a content field to capture the actual record. Each record's content may have a different inner structure. A structural RepInfo must be added to the record to interpret its inner structure. See figure 16 for the UML diagram of this interface.

7.11 PDSREPINFO INTERFACE

PDSRepInfo interface defines a set of PDSRepInfoRecords, each record describes a single RepInfo.

The set of RepInfos may be used to interpret the Content Data or a PDI record of the AIP. RepInfo records for the Content Data may be added during ingest, or later using the addContetDataRepInfo() API.

Changes to the RepInfo may cause the generation of a new version to the AIP. See figure 16 for the UML diagram of this interface.

7.12 PDSREPINFORECORD INTERFACE

The PDSRepInfoRecord contains a link to the content of the RepInfo and a reference to a PDSRepInfo interface. The latter describes the RepInfo of this RepInfo's content, thus creating the RepInfo network. If the RepInfo's content does not have RepInfo of its own, its RepInfo reference must be null. The RepInfo record may have different types indicating what kind of RepInfo it represents: "Semantic", "Structural" etc.

The RepInfo record also contains a short description of the RepInfo it represents. The description may be "Word 5.0 executable", "PDF specification" etc. This provides the user with some knowledge of what sort of RepInfo this record points at without having to access the object that actually contains that RepInfo. See figure 16 for the UML diagram of this interface.

7.13 PDSAIPID INTERFACE

PDSAipId interface represents a full ID for an AIP.

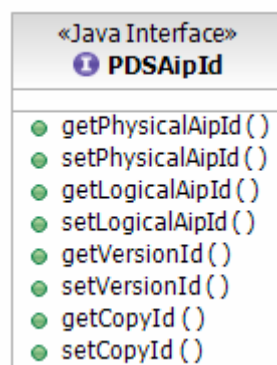


Figure 17: PDSAipId interface

The AIP ID is constructed of three parts:

1. logical ID - a globally unique ID that is shared among all versions and copies of an AIP
2. version ID - a globally unique ID that identifies the version of this AIP among the various version originated from the same AIP





3. copy ID - a globally unique ID that identifies the copy of this AIP among the various identical copies of the AIP

This interface enables access to each part of the AIP ID, as well as to the complete ID, the physical AIP ID.

See figure 17 for the UML diagram of this interface.

7.14 PDSAIPPOLICY INTERFACE

PDSAipPolicy interface defines the specific policies for the preservation of an AIP.

This interface includes a number of policies. Among them are:

- StoreOriginal - All AIPs in the PDS are preserved in PDS' own unpacked format; AIPs submitted in a different ("packed") format are converted to PDS' unpacked format upon ingest. The 'StoreOriginal' policy makes PDS preserve, in addition to the unpacked AIP, the original packed AIP in the original format used.
- FixityAlgorithm - indicates which Fixity algorithm should be used for this AIP
- AipImportance - indicates the importance of this AIP. Legal values are:
 - High
 - Medium
 - Low

This parameter may affect the number of copies made for this specific AIP.

- Deletable – indicates whether this AIP may be deleted in the future

These policies are required at ingest call, but may also be changed later. Note that some policies cannot be changed after ingest. For instance, if an AIP was ingested with false value for storeOriginal, it cannot be changed later to true as the packed AIP is not available anymore.

See figure 18 for the UML diagram of this interface.

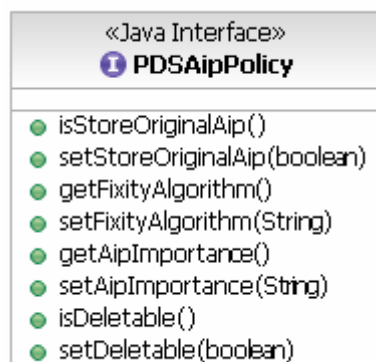


Figure 18: PDSAipPolicy interface





7.15 PDSPOLICY INTERFACE

PDSPolicy interface represents PDS policies.

7.16 PDSLINK INTERFACES

PDSLink represents different types of reference links. It may be a URL, a file path, DOI and any other type of reference PDS supports. PDSLink is used in all PDS interfaces that use reference links.

The following types of links are supported by PDS:

- URL
- DOI
- AipId
- CPID
- PATH
- AipMemRef

The list of supported link types may be retrieved by calling the `getSupportedLinkTypes()` API. `AipMemRef` link type requires a link value implementation of `PDSInternalLinkValue` as it refers to an in-memory link type. All other link types require `PDSEternalLinkValue` that will be represented by a byte array. In the future `RepInfo` field may be added to the reference link type to describe it.

In addition to the type and value of the link, we enable a finer grained representation of the referenced object by using the following fields:

- `beginOffset` - defines the exact starting position within the link value, the PDS link refers to. The default value is 0.
- `length` - defines how many bytes, starting from `beginOffset`, are referred to by the PDS link.
- `completeRange` - if this flag is set, the `beginOffset` and `length` fields are ignored and the complete available range is referenced. The default value of this flag is 'true'.

`PDSLinkValue` interface provides an abstraction to different types of link values. It is a marker interface that is used for valid types of link values. Currently there are two valid types - a byte array represented by the `PDSEternalLinkValue` extension and a memory reference to `PDSAip` that is represented by the `PDSInternalLinkValue` extension.

`PDSInternalLinkValue` interface provides an extension to `PDSLinkValue` and meant for link values that are internal to the process memory. It represents a link to an in-memory `PDSAip` implementation.

`PDSEternalLinkValue` interface provides an extension to `PDSLinkValue` and meant for link values that are external to the process memory. It represents different types of external reference links supported by PDS. It uses a byte array to get and set the link value. Any link that is not in-memory link is considered an external link and this interface should be used to store its value.





See figure 19 for the PDSLink class UML diagram and its sub-classes.

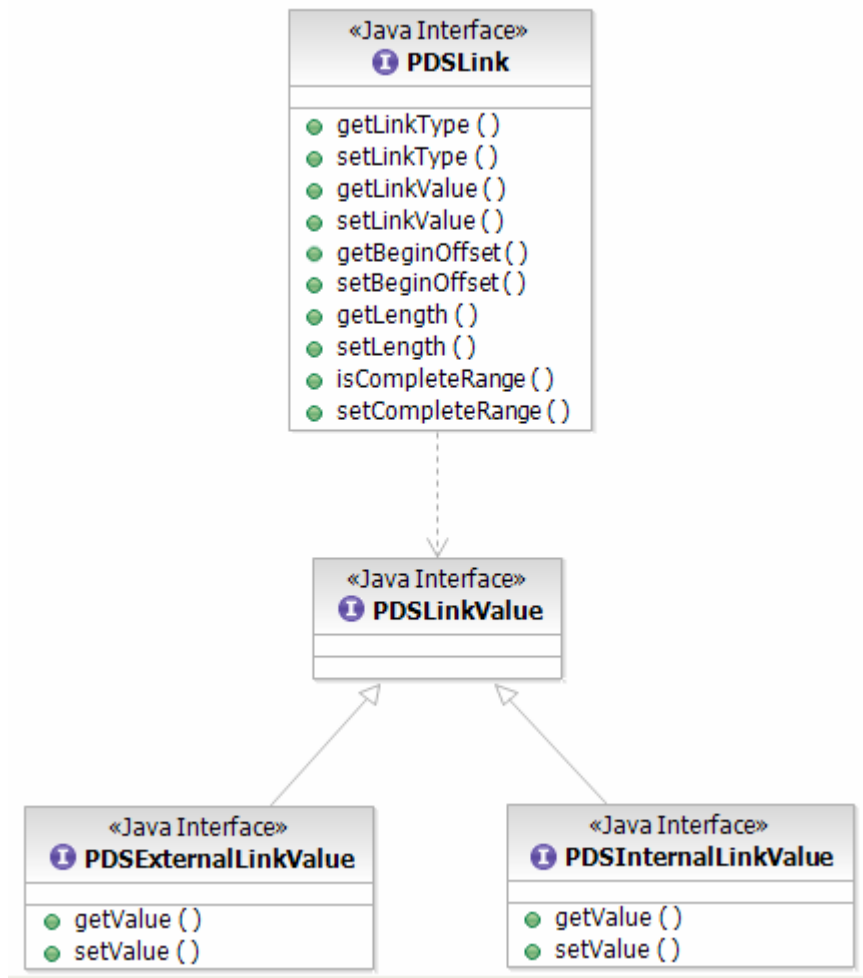


Figure 19: PDSLink interface and sub-interfaces

7.17 PDSPACKAGINGHANDLER INTERFACES

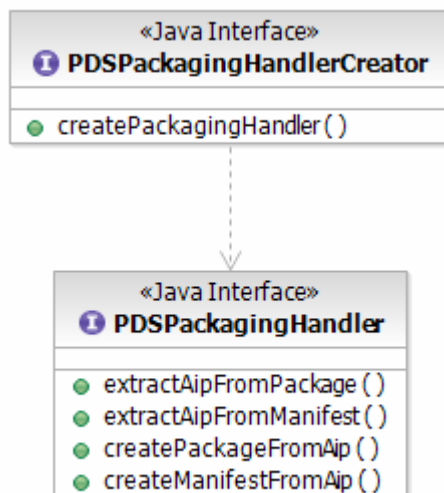


Figure 20: PDSPackagingHandler and PDSPackagingHandlerCreator interfaces





PDSPackagingHandler interface provides the definition of the methods the packaging implementation must provide to the Preservation Engine in order to pack and unpack AIPs according to a specific format.

PDSPackagingHandlerCreator is an interface for creating a packaging handler.

See figure 20 for the UML diagram of these interfaces.





8 DATA FLOW

This section describes the data flow in three principal scenarios in the preservation system: ingestion of an AIP, access of an AIP, and migration of an AIP. For simplicity, we'll assume that each AIP is an AIU i.e. it's an atomic AIP and doesn't embed additional AIPs inside it. For each scenario, we describe the data flow within the PDS Preservation Engine. Note that this activity causes the Preservation Engine to generate multiple XSets in the XAM Library component which in turn will go through the XAM to OSD and HL OSD components, which in turn will generate multiple user objects in Object Store. However, the flow from the Preservation Engine to XAM Library, XAM to OSD, HL OSD, and OSD is not described here.

8.1 INGEST AIP

1. The Preservation DataStore ingest function is called with an AIP. To support flexibility, the Preservation Engine will either accept a packaged AIP or the various parts of the AIP (content data object, RepInfo, context, provenance, etc) that comprises the AIP. In the latter case, the Preservation Engine will package itself the various parts into an AIP.
2. The Reference Manager either assigns or validates the given Persistent Globally Unique ID for the AIP. The AIP ID also resides in the OAIS Data Management entity or CASPAR Directory Service.
3. The RepInfo Manager validates and fetches some of the RepInfo network of the content data object.
4. The Fixity Manager computes the fixity for this AIP.
5. The Provenance Manager generates the provenance structure and adds the initial events (events that occurred before and during the ingest).
6. The Context Manager checks the context referential integrity.
7. The RepInfo Manager associates the PDI RepInfo with the given AIP.
8. The Placement Manager computes to which cluster to assign the given AIP.
9. The Placement Manager updates the cluster object and optionally compresses it. It also updates the AIP-to-MediaUnit table.
10. The Placement Manager optionally exports (unmount) the cluster object to the media unit using a self-describing self-contained format.
11. Preservation DataStore returns the AIP ID to the caller.

8.2 ACCESS AIP

1. The Preservation DataStore access function is called with an AIP ID.
2. The Placement Manager uses the persistent AIP-to-MediaUnit mapping table (catalog) to get the media unit of the given AIP.
3. The Placement Manager mounts the adequate media unit, decompresses it if was compressed and retrieves the given AIP.
4. The Placement Manager fetches the Content Information RepInfo from the media unit.
5. The RepInfo Manager inserts the relevant portion of the RepInfo network into the AIP.
6. The Placement Manager fetches the PDI RepInfo.
7. The RepInfo Manager inserts the relevant portion of the PDI RepInfo into the AIP.
8. The Fixity Manager validates the fixity.
9. The Provenance Manager adds the access event.
10. Preservation DataStore puts for the caller the copy of the required AIP or the requested part of the AIP in a staging storage.
11. Preservation DataStore returns to the caller with the link to the staging area.





8.3 MIGRATE AIP

1. The Preservation Engine in the old system performs the AIP access flow described in section above.
2. The Migration Manager supports transformations in one of the following ways:
 - a. Provide a storlet container that can run the transformation
 - b. Provide a list of the applications (RepInfos) that are in the old system. The administrator will map those to the new applications and then the Migration Manager will provide scripts to perform those transformations
3. The Fixity Manager computes the fixity if a transformation was applied.
4. The Context Manager validates that the links are valid and updates the context.
5. The Provenance Manager adds the migration events.
6. The Migration Manager generates a self-describing self-contained AIP and exports it from the old preservation system.
 - a. For optimization reasons, we may want to export a complete cluster at a time instead of an AIP at a time.
7. The Migration Manager in the new preservation system performs the Ingest AIP flow described above.





9 AIP EXAMPLE: MAPPING A SAFE MANIFEST FILE TO AIP

This section demonstrates PDS with a specific type of scientific data packaged in the XFDU/SAFE format. In this section, we show how this data is mapped to AIP stored in PDS using an example XFDU/SAFE manifest file attached in the appendix. In the next section, we describe a tool to assist in generating AIPs for that data packaged with XFDU/SAFE.

9.1 GENERAL DESCRIPTION

Each SAFE product consists of a *SAFE content unit*, which contains one or more *content units*.

If a SAFE product contains more than one content unit, it should be mapped to a single AIC containing an AIU for each content unit of the SAFE product.

The AIP information shall be extracted from the manifest file. Some of the ingested metadata are common to all content units, and should therefore be kept as AIC metadata; some are specific to one of the content units should be kept in the respective AIUs:

- Content data, RepInfo and fixity information are specific for each content unit in a SAFE product. They should be stored separately for each AIU.
- Context information and provenance information are product-wide information and are common to all content units within a SAFE product. They should be stored for the whole AIC.

Note: A SAFE product may also be viewed as an AIU, not an AIC. The content units in a product may therefore be regarded as "Archive Information Sub-units" within the same AIU. This is to stress that the connection between the data of a content unit and the metadata describing them must be maintained.

9.2 SUGGESTED MAPPING

In the following table, we present a mapping of objects and types in a SAFE manifest to OAIS-compliant types. We used ER01_SAR_IM__OP_19950630T152126_19950630T152235_ACS_20697_50E0.SAFE (see appendix A) as a sample. Please note that this is a preliminary mapping which may change in the future.

Note: the following table treats a SAFE product as if it were an AIC. If you consider a SAFE product to be an AIU, simply replace all references to "AIU" with "Archive Information Sub-unit" and all references to "AIC" with "AIU".

Table 3: Mapping SAFE manifest file to AIP

Type in SAFE	XPath in manifest file	value	OAIS-compliant type according to our suggested mapping
Syntactic RepInfo file	<code>/*/metadataObject[classification=SYNTAX && category=REP]/metadataReference/xlink:href</code>	index.xsd	RepInfo of index AIU
Metadata Reference mimeType	<code>/*/metadataObject[classification=SYNTAX && category=REP]/metadataReference/mimeType</code>	text/xml	RepInfo of RepInfo of index AIU
Syntactic	<code>/*/metadataObject[classificati</code>	measurement.xsd	RepInfo of measurement AIU





Type in SAFE	XPath in manifest file	value	OAIS-compliant type according to our suggested mapping
RepInfo file	on=SYNTAX && category=REP]/metadataReference/xlink:href		
Metadata Reference mimeType	/*/metadataObject[classification=SYNTAX && category=REP]/metadataReference/mimeType	text/xml	RepInfo of RepInfo of measurement AIU
Product type	XFDU/version	esa/safe/1.0/ers/ami/sar/level-0	context of the AIC
Referenced schema	XFDU/xmlns	http://www.w3.org/2001/XMLSchema	Packaging Info – do not store
Referenced schema	XFDU/xmlns	http://www.w3.org/1999/xlink	Packaging Info – do not store
Referenced schema	XFDU/xmlns	http://www.ccsds.org/xfdu/2004	Packaging Info – do not store
DMD	/*/metadataObject[ID=platform && classification=DESCRIPTION && category=DMD]/metadataWrap/xmlData (*)	platform...	context of the AIC
MetadataWrap mimeType	/*/metadataObject[ID=platform && classification=DESCRIPTION && category=DMD]/metadataWrap/mimeType	text/xml	RepInfo of context Note: each context item has such RepInfo
DMD	/*/metadataObject[ID=measurementQualityInformation && classification=DESCRIPTION && category=DMD]/metadataWrap/xmlData	qualityInformation ...	context of the AIC
DMD	/*/metadataObject[ID=acquisitionPeriod && classification=DESCRIPTION && category=DMD]/metadataWrap/xmlData	acquisitionPeriod ...	context of the AIC
DMD	/*/metadataObject[ID=measurementOrbitReference && classification=DESCRIPTION && category=DMD]/metadataWrap/xmlData	orbitReference ...	context of the AIC





Type in SAFE	XPath in manifest file	value	OAIS-compliant type according to our suggested mapping
PDI	/*/metadataObject[ID=processing && classification=PROVENANCE && category=PDI]/metadataWrap/xmlData	processing...	Ingested provenance of the AIC. Should be saved in one piece as opaque provenance data (with the appropriate RepInfo) along with breaking it to events and saving it in our provenance format (with our RepInfo). In this case 3 events.
Content unit #1			
dataObject file	/*/dataObject[ID=measurementIndexData]/byteStream/fileLocation/xlink:href	index.dat	Content data of index AIU.
RepInfo for dataObject	/*/dataObject[ID=measurementIndexData]/repID	measurementIndex Schema	Packaging Info – do not store
dataObject checksum	/*/dataObject[ID=measurementIndexData]/byteStream/checksum	2ab9c3cb64602585f0152606c7b767b0	Ingested fixity of index AIU (computed on content data only – in this case, index.dat) used to verify ingested data integrity and also documented as 1 st fixity event.
dataObject checksum type	/*/dataObject[ID=measurementIndexData]/byteStream/checksumType	MD5	Part of the index AIU ingested fixity. Saved as Fixity RepInfo. The PDS computes this fixity algorithm on the ingested content data to verify data integrity.
byteStream mimeType	/*/dataObject[ID=measurementIndexData]/byteStream/mimeType	application/octetstream	Type of the content data
Content unit #2			
dataObject file	/*/dataObject[ID=measurementData]/byteStream/fileLocation/xlink:href	measurement.dat	Content data of measurement AIU.
RepInfo for dataObject	/*/dataObject[ID=measurementData]/repID	measurementSchema	Packaging Info – do not store
dataObject checksum	/*/dataObject[ID=measurementData]/byteStream/checksum	83ca293826ce38622f84082b68365d8b	Ingested fixity of measurement AIP (computed on content data only, in this case, measurement.dat) used to verify ingested data integrity and also documented as 1 st fixity event.
dataObject checksum type	/*/dataObject[ID=measurementData]/byteStream/checksumType	MD5	Part of the measurement AIU ingested fixity. Saved as Fixity RepInfo. The PDS computes





Type in SAFE	XPath in manifest file	value	OAIS-compliant type according to our suggested mapping
			this fixity algorithm on the ingested content data to verify data integrity.
byteStream mimeType	/*/dataObject[ID=measurementIndexData]/ bytestream/mimeType	application/octetstream	Type of the content data





10 XFDU AIP GENERATOR

The XFDU AIP Generator (XAG) is an easy-to-use graphical tool for viewing, creating and editing manifest files, the AIP's "table of contents". The tool is created using EMF (Eclipse Modeling Framework), an Eclipse plug-in enabling the creation and further manipulation of a selected model – in our case, a model based on the XFDU schema.

10.1 ADVANTAGES OF XAG

XAG provides an easy-to-use graphical interface for creating and editing XFDU manifest files. Like other XML editors, it represents the contents of the XML file in a graphic table or tree. However, XAG has a major advantage over other graphical XML editors: it allows only the usage of valid elements and values. Unlike regular XML editors, in which validation against the schema takes place only after the XML file has been written, XAG enforces the XFDU schema rules as you go. This means that whenever one wishes to add an element to the XML file, he or she selects it from a list that contains only elements allowed by the XFDU schema; whenever one wishes to set a new value to a certain property, he or she can only select or enter valid values for the property, etc. Removing confusing, irrelevant options makes it easier for the user to work – and significantly reduces errors.

After singing its praise, it is important to point out that XAG can only create and edit the *manifest file* of an XFDU-based AIP – not the entire AIP. Ingesting actual data and to such an AIP, as well as extracting data from it, must still be done using other tools (PDS web interface, for example). However, future improvements may allow a smoother integration of XAG with such tools.

10.2 INSTRUCTIONS FOR USE

XAG uses the standard Microsoft Windows GUI. This means that the **File** and **Edit** command menus in XAG are very similar to those in any other Windows application. The regular windows shortcuts (**CTRL+C** to copy, **CTRL+S** to save, **CTRL+Z** to undo, etc.) will also work in XAG. The following figure is an overview of XAG.



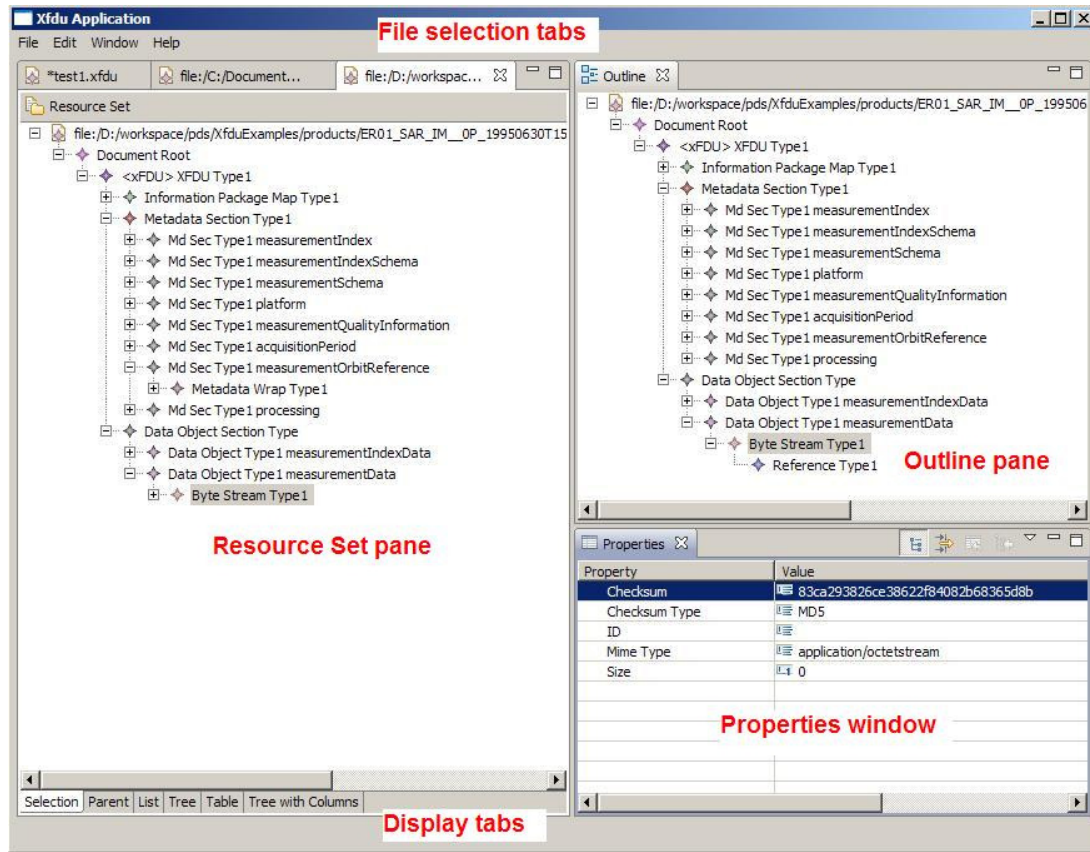


Figure 21: XAG overview

Create a new XFDU manifest file:

1. Launch XAG.
2. Select **File | New... | xfd� model...** A dialog box will open.
3. Name the new file (it must end in .safe).
4. From the **Model Object** combo box, select **XFDU**. Click **Finish**.

Edit an existing file

1. Select **File | Open...** A dialog box will open.
2. Select the desired .safe file. A new tab will appear on the top of the “Resource Set” pane.

Add an item

1. Right click an element on either the “Resource Set” pane or the “Outline pane”.
2. From the right-click menu, select **New Child** or **New Sibling**.
3. Select the type of the element you wish to add. The new element will appear on both panes.

Note: the addition menu is *context-sensitive*, meaning it will only display the types that the schema allows to add under (or on side of) the element you have right-clicked on. For example, you may wish to start a new manifest file by creating its Metadata Section, as shown below:



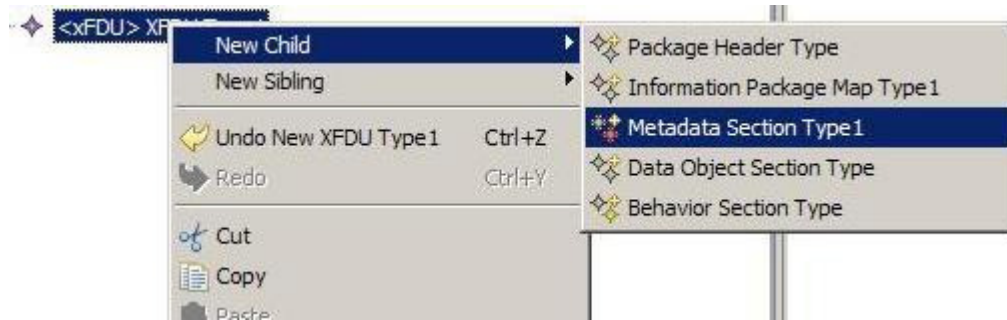


Figure 22: Addition menu with metadata section available

The SAFE XFDU schema allows only one element of type Metadata Section. To enforce this restriction and prevent the creation of an invalid manifest file, XAG grays out the Metadata Section entry on the addition menu after such an element has been chosen, as the following figure shows:

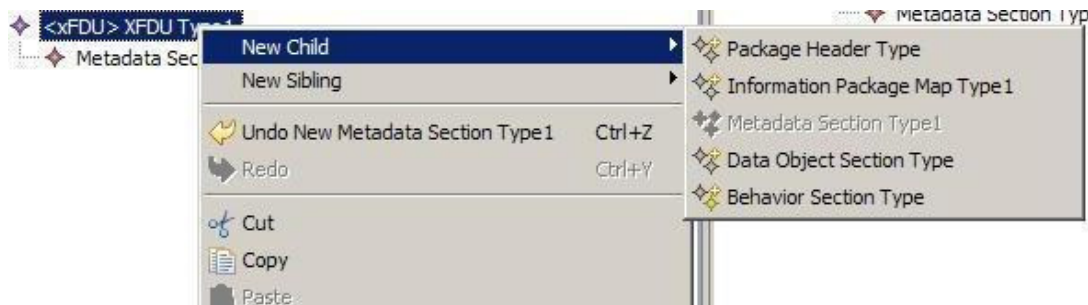


Figure 23: Addition menu with metadata section grayed out

Edit the properties of an item

1. From the "Resource Set" pane or the "Outline" pane, select the item whose properties you wish to edit. These will appear in the "Properties" window.
2. Click the "value" column of the desired property; the value will be highlighted:
 - a. If a small down-arrow appears on the right, click it and select the desired value from the drop-down menu. Note that the menu only contains values valid for this property.
 - b. Otherwise, type in the new value.

Delete an item

1. Right-click the item you'd like delete. A menu will appear.
2. From the menu, select **Delete**.





11 REFERENCES

- [1] M. Factor, D. Naor, S. Rabinovici-Cohen, L. Ramati, P. Reshef, and J. Satran. "The Need for Preservation Aware Storage - A Position Paper". ACM SIGOPS Operating Systems Review, Special Issue on File and Storage Systems, Volume 41, Issue 1, pages 19-23, (2007)
- [2] M. Factor, D. Naor, S. Rabinovici-Cohen, L. Ramati, P. Reshef, J. Satran, and D.L Giaretta. "Preservation DataStores: Architecture for Preservation Aware Storage", In Proc. IEEE Conference on Mass Storage Systems and Technologies (MSST), San Diego, USA (2007)
- [3] "Towards OAIS-Based Preservation Aware Storage - A White Paper". See <http://www.haifa.il.ibm.com/projects/storage/datastores/public.html>
- [4] ISO 14721:2003, Blue Book. Issue 1. CCSDS, 650.0-B-1: Reference Model for an Open Archival, Information System (OAIS), (2002)
- [5] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. a position paper. In Local to Global Data Interoperability – Challenges and Technologies, Sardinia Italy., pages 119–123, June 2005.
- [6] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In Proceedings of the 2006 USENIX Annual Technical Conference, June 2006.
- [7] SNIA - Networking Industry Association, Data Management Group, XAM (Extensible Access Method). See <http://www.snia-dmf.org/xam/>
- [8] SNIA - Storage Networking Industry Association. OSD: Object Based Storage Devices Technical Work Group.
- [9] International Committee for Information Technology Standards (formerly NCITS), SCSI Object-Based Storage Device Commands (OSD). Document Number: ANSI/INCITS 400-2004, technical editor: R.O. Weber, December 2004.
- [10] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. "Dynamic Function Placement for Data-intensive Cluster Computing". In *Proc. of 2000 USENIX Annual Technical Conference*, San Diego, June 2000.
- [11] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle, "Active Disks for Large-Scale Data Processing". In *IEEE Computer*, June 2001
- [12] Reagan W. Moore and Richard Marciano. "Building preservation environments". In *JCDL '05: Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, pages 424–424, New York, NY, USA, 2005. ACM Press.
- [13] Reagan W. Moore, Joseph F. JaJa, and Robert Chadduck. "Mitigating risk of data loss in preservation environments". In *MSST '05: 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 39–48, Washington, DC, USA, 2005. IEEE Computer Society.
- [14] SRB – Storage Resource Broker. See http://www.sdsc.edu/srb/index.php/Main_Page
- [15] iRODS - Intelligent Rule-Oriented Data management System. See http://irods.sdsc.edu/index.php/Main_Page





APPENDIX A – SAFE PRODUCT MANIFEST FILE

Following is the content of the manifest file of the SAFE product - ER01_SAR_IM_OP_19950630T152126_19950630T152235_ACS_20697_50E0.SAFE which we used in the example in section 10:

```
<?xml version="1.0" encoding="UTF-8"?>

<xfdu:XFDU version="esa/safe/1.0/ers/ami/sar/level-0"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xfdu="http://www.ccsds.org/xfdu/2004"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ccsds.org/xfdu/2004
../schemas/int/esa/safe/1.0/xfdu.xsd">
  <informationPackageMap>
    <xfdu:contentUnit unitType="SAFE Archive Information Package"
textInfo="SAFE Archive Information Package" dmdID="platform
acquisitionPeriod" pdiID="processing">
      <xfdu:contentUnit unitType="Measurement Data Index"
textInfo="Measurement Data Index">
        <dataObjectPointer dataObjectID="measurementIndexData"/>
      </xfdu:contentUnit>
      <xfdu:contentUnit unitType="Measurement Data Unit"
textInfo="Measurement Data Unit" repID="measurementSchema"
dmdID="measurementOrbitReference measurementIndex
measurementQualityInformation">
        <dataObjectPointer dataObjectID="measurementData"/>
      </xfdu:contentUnit>
    </xfdu:contentUnit>
  </informationPackageMap>
  <metadataSection>
    <metadataObject ID="measurementIndex" classification="DESCRIPTION"
category="DMD">
      <dataObjectPointer dataObjectID="measurementIndexData"/>
    </metadataObject>

    <metadataObject ID="measurementIndexSchema" classification="SYNTAX"
category="REP">
      <metadataReference mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SDF" locType="URL" xlink:href="index.xsd"/>
    </metadataObject>
    <metadataObject ID="measurementSchema" classification="SYNTAX"
category="REP">
      <metadataReference mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SDF" locType="URL" xlink:href="measurement.xsd"/>
    </metadataObject>
    <metadataObject ID="platform" classification="DESCRIPTION"
category="DMD">
      <metadataWrap mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SAFE" textInfo="Platform Description">
        <xmlData>
          <platform xmlns="http://www.esa.int/safe/1.0">
            <nssdcIdentifier>1991-050A</nssdcIdentifier>
            <familyName>ERS</familyName>
            <number>1</number>
            <instrument>
              <familyName abbreviation="AMI">Active Microwave
Instrument</familyName>
              <number>1</number>
            </instrument>
          </platform>
        </xmlData>
      </metadataWrap>
    </metadataObject>
  </metadataSection>
</xfdu:XFDU>
```





```

        </instrument>
        <timeReference>
            <utc>1995-06-30T14:50:15.809999Z</utc>
            <clock>1893150196</clock>
            <clockStep>3906250</clockStep>
        </timeReference>
    </platform>
</xmlData>
</metadataWrap>
</metadataObject>
<metadataObject ID="measurementQualityInformation"
classification="DESCRIPTION" category="DMD">
    <metadataWrap mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SAFE" textInfo="Quality Information">
    <xmlData>
        <qualityInformation xmlns="http://www.esa.int/safe/1.0">
            <corruptedElements>
                <location preceding="3">line</location>
                <count>3</count>
                <evidence type="TIME-SPIKE"/>
            </corruptedElements>
            <corruptedElements>
                <location following="4">line</location>
                <count>8</count>
                <evidence type="TIME-SPIKE"/>
            </corruptedElements>
            <corruptedElements>
                <location following="117232">line</location>
                <count>1</count>
                <evidence type="TIME-SPIKE"/>
            </corruptedElements>
            <corruptedElements>
                <location following="124214">line</location>
                <count>1</count>
                <evidence type="TIME-SPIKE"/>
            </corruptedElements>
            <corruptedElements>
                <location following="128206">line</location>
                <count>1</count>
                <evidence type="TIME-SPIKE"/>
            </corruptedElements>
            <corruptedElements>
                <location following="132307">line</location>
                <count>1</count>
                <evidence type="TIME-SPIKE"/>
            </corruptedElements>
        </qualityInformation>
    </xmlData>
</metadataWrap>
</metadataObject>
<metadataObject ID="acquisitionPeriod" classification="DESCRIPTION"
category="DMD">
    <metadataWrap mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SAFE" textInfo="Acquisition Period">
    <xmlData>
        <acquisitionPeriod xmlns="http://www.esa.int/safe/1.0">
            <startTime>1995-06-30T15:21:26.248Z</startTime>
            <stopTime>1995-06-30T15:22:34.197Z</stopTime>
        </acquisitionPeriod>
    </xmlData>

```





```

    </metadataWrap>
  </metadataObject>
  <metadataObject ID="measurementOrbitReference"
classification="DESCRIPTION" category="DMD">
    <metadataWrap mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SAFE" textInfo="Orbit Reference">
      <xmlData>
        <orbitReference xmlns="http://www.esa.int/safe/1.0">
          <orbitNumber type="start">20697</orbitNumber>
          <orbitNumber type="stop">20697</orbitNumber>
          <cycleNumber>147</cycleNumber>
          <phaseIdentifier>G</phaseIdentifier>
        </orbitReference>
      </xmlData>
    </metadataWrap>
  </metadataObject>
  <metadataObject ID="processing" classification="PROVENANCE"
category="PDI">
    <metadataWrap mimeType="text/xml" vocabularyMdType="OTHER"
otherMdType="SAFE" textInfo="Processing">
      <xmlData>
        <processing name="Conversion" start="2007-01-
25T15:20:42.566Z" stop="2007-01-25T15:24:55.479Z"
xmlns="http://www.esa.int/safe/1.0">
          <facility country="Italy" name="ACS" organisation="ESA-
ESRIN" site="Rome"/>
          <software name="Wilma2SafeConverter" version=""/>
          <resource
name="ER01_SAR_IM__0P_19950630T152126_19950630T152234_CUB_20697.WILM"
href="" role="Transcribed product">
            <processing name="Transcription" start="2000-04-
21T08:53:22Z" stop="2000-04-21T08:56:48Z">
              <facility country="" name="ESR" organisation=""
site=""/>
              <resource name="UNKNOWN" href="" role="Downlinked
telemetry">
                <processing name="Acquisition" start="1995-06-
30T15:21:26.248Z" stop="1995-06-30T15:22:34.197Z">
                  <facility country="Ecuador" name="CPE"
organisation="ESA-ESRIN" site="Cotopaxi"/>
                </processing>
              </resource>
            </processing>
          </resource>
        </processing>
      </xmlData>
    </metadataWrap>
  </metadataObject>
</metadataSection>
<dataObjectSection>
  <dataObject repID="measurementIndexSchema" ID="measurementIndexData">
    <byteStream mimeType="application/octetstream"
checksum="2ab9c3cb64602585f0152606c7b767b0" checksumType="MD5" size="0">
      <fileLocation locType="URL" textInfo="Measurement Data Index"
xlink:href="index.dat"/>
    </byteStream>
  </dataObject>
  <dataObject repID="measurementSchema" ID="measurementData">
    <byteStream mimeType="application/octetstream"
checksum="83ca293826ce38622f84082b68365d8b" checksumType="MD5" size="0">

```





```
<fileLocation locType="URL" textInfo="Measurement Data Unit"
xlink:href="measurement.dat"/>
</byteStream>
</dataObject>
</dataObjectSection>
</xfdu:XFDU>
```

