Digital Object Prototypes: An Effective Realization of Digital Object Types

Kostas Saidis[1], George Pyrounakis[2], Mara Nikolaidou[2], and Alex Delis[1]

[1] Dept. of Informatics and Telecommunications
[2] Libraries Computer Center
University of Athens
University Campus, Athens, 157 84, Greece
{saiko, forky, mara, ad}@di.uoa.gr

Abstract. Digital Object Prototypes (DOPs) provide the DL designer with the ability to model diverse types of digital objects in a uniform manner while offering digital object type conformance; objects conform to the designer's type definitions automatically. In this paper, we outline how DOPs effectively capture and express digital object typing information and finally assist in the development of unified web-based DL services such as adaptive cataloguing, batch digital object ingestion and automatic digital content conversions. In contrast, conventional DL services require custom implementations for each different type of material.

1 Introduction

Several formats and standards, including METS [10], MPEG-21 [15], FOXML [7] and RDF [11] are in general able to encode heterogeneous content. What they all have in common is their ability to store and retrieve arbitrary specializations of a digital object's constituent components, namely, files, metadata, behaviors and relationships [9]. The derived digital object typing information – that is, which components constitute each different type of object and how each object behaves – is not realized in a manner suitable for effective use by higher level DL application logic including DL modules and services [13].

Our main objective is to enhance how we express and use the types of digital objects independently of their low-level encoding format used for storage. Digital object prototypes (DOPs) [13] provide a mechanism that uniformly resolves digital object typing issues in an automated manner. The latter releases DL users such as cataloguers, developers and designers from dealing with the underlying complexity of typing manually. A DOP is a *digital object type definition* that provides a detailed specification of its constituent parts and behaviors. Digital objects are conceived as instances of their respective prototypes. DOPs enable the generation of *user-defined types* of digital objects, allowing the DL designer to model the specialities of each type of object in a fine-grained manner, while offering an implementation that guarantees that all objects conform to their type automatically. Using DOPs, the addition of a new digital object type requires no custom development and services can be developed to operate directly on all types of material without additional coding for handling "special" cases.

DOPs assist in dealing with important "every day" DL development issues in a unified way: how to speed up and simplify cataloguing, how to automate content ingestion, how to develop effective web interfaces for presenting and manipulating heterogeneous types of digital objects. In this paper, we focus on the benefits offered by the deployment of DOPs in the development of high level services in Pergamos, the University of Athens DL. In particular, we point out how web based services such as browsing, cataloguing, batch ingestion and automatic digital content conversion cope with any type of DOP defined object, while having all services reside in a single, uniform implementation.

The remainder of the paper is organized as follows. Section 2 provides a detailed description of the current implementation of DOPs and pinpoints how DOPs assist on the development of uniform yet effective DL services. In Section 3 we present several DOP examples originating from Pergamos collections. Finally, Section 4 concludes the paper discussing related and future work.

2 Digital Object Prototypes in Pergamos

We have implemented DOPs in Java. As depicted in Figure 1a, DOPs operate atop the repository / storage layer of the DL (in Pergamos we use FEDORA [14]).

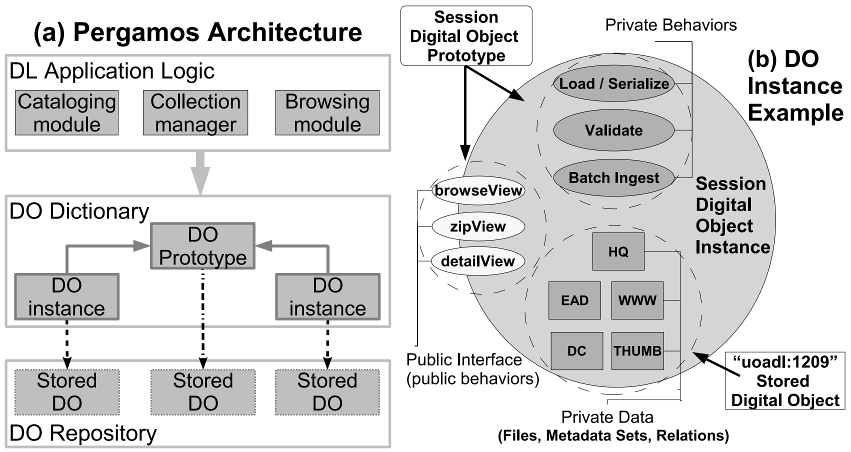

Fig. 1. (a) The 3-tier Pergamos architecture incorporating the "type enforcement" layer of DO Dictionary [13] and (b) A digital object instance as composed by its respective prototype and the underlying stored digital object

The *DO Dictionary* layer of Figure 1a exposes the DOPs API to high level DL services or the application logic. The underlying repository's "mechanics" remain hidden, since all service functionality is directed through DOPs. We define DOPs in terms of XML documents, that are loaded by the *DO Dictionary* at bootstrap time. These XML documents provide the type specification that is translated to a Java representation wrapped by the DOPs API. At runtime, the *DO Dictionary* loads stored digital objects from the repository and generates Java artifacts named digital object instances that conform to their respective DOP definition. High level services operate on digital object instances; any modification occurring in an instance's data is serialized back to the repository when the object is saved.

In order to illustrate how DOPs effectively realize digital object types, in this section we use examples drawn from the Senate Archive's Session Proceedings collection in Pergamos DL. We model Session Proceedings using `Session` and `Page` DOPs; each Senate `Session` is modelled as a complex object containing `Pages`. Figure 1b depicts the runtime representation of a `Session` digital object instance, while Figure 2 illustrates the definition of the `Session` DOP, encoded in XML. The `Session` instance reflects the specifications found in the `Session` DOP. The instance's behaviors are defined in the DOP the instance conforms to, while its metadata, files and relations are loaded from its underlying stored digital object.

```xml
<prototype id="Session">
<MDSets><!-- Metadata definition -->
 <MDSet id="dc">
  <label lang="en">Dublin Core Metadata</label>
  <datastream id="DC" MDType="descriptive"
     loader="gr.uoa.dl.core.xml.StandardLoader"
     serializer="gr.uoa.dl.core.xml.DCSerializer"/>
  <fields>
   <field id="dc:date" isMandatory="true"
     isRepeatable="false" isHidden="false"
     validation ="gr.uoa.dl.core.validation.DateFormat">
    <label lang="en">Date</label>
   </field>
   <field id="dc:identifier_physical" isMandatory="true"
     isRepeatable="false" isHidden="true">
    <label lang="en">Call number</label>
   </field>
   ...
  </fields>
 </MDSet>
 <MDSet id="ead">
  <label lang="en">EAD like Metadata</label>
  <datastream id="EAD" MDType="descriptive"
     loader="gr.uoa.dl.core.xml.StandardLoader"
     serializer="gr.uoa.dl.core.xml.EADSerializer"/>
  <fields>
   <field id="did_unitid"/>
   ...
  </fields>
 </MDSet>
 <mappings>
  <mapping id="identifier">
   <from="ead.did_unitid">
   <to="dc.dc:identifier_physical"/>
  </mapping>
 </mappings>
</MDSets>
```

```xml
<files><!-- Files definition -->
 <file id="zip" type="container" datastream="ZIP">
  <label lang="en">ZIP file</label>
  <mime-type id="application/zip"/>
  <batchIngest targetTypeId="page" targetFileId="hq"/>
 </file>
</files>
<relations><!-- Relationships definition -->
    <structure>
        <childType>Page</childType>
    </structure>
    <references allowCustomURL="false" allowCustomDO="true"/>
</relations>
<behaviours><!-- Behaviours definition -->
 <schemes>
  <scheme id="browseView" isDefault="true">
   <label lang="en">Short View</label>
   <element id="MDSets.dc.dc:identifier"/>
   <element id="MDSets.dc.dc:title"/>
   <element id="MDSets.dc.dc:date"/>
  </scheme>
  <scheme id="zipView">
   <label lang="en">Short View</label>
   <element id="MDSets.dc.dc:title"/>
   <element id="files.zip"/>
  </scheme>
  <scheme id="detailView">
   <label lang="en">Detail View</label>
   <element id="MDSets.dc.dc:identifier"/>
   <element id="MDSets.dc.dc:identifier_physical"/>
   <element id="MDSets.dc.dc:title"/>
   <element id="MDSets.dc.dc:date"/>
   ...
  </scheme>
 </schemes>
</behaviours>
</prototype>
```

Fig. 2. The `Session` prototype defined in XML terms

DOP definitions are encoded in XML as depicted by the `Session` DOP of Figure 2 and are made up of four parts according to [9]: (a) metadata element set definitions expressed in the `MDSets` XML section, (b) digital content specifications expressed in the `files` section, (c) relationships, defined in the `relations` section and (d) behaviors, defined in the `behaviors` XML section. In the following we provide a detailed description of each of these four definition parts, while, in parallel, we discuss how these type definitions are interpreted at runtime. It is worth pointing out that, although most of the examples we use herein originate from object input scenarios, the automatic type conformance offered by DOPs covers all aspects of digital object manipulation. The DOPs framework is not a static digital object model. On the contrary, it can be conceived as a framework that allows users to define their own digital object models.

2.1 Behaviors in DOPs

The behaviors of a digital object constitute the set of operations it supports. All the instances of the same DOP share the same behaviors; for example, all Session Proceedings behave in the same manner. This is reflected by the fact that with DOPs, behaviors are defined only in the object's respective prototype and are automatically bound to the digital object instance at runtime by the *DO Dictionary*.

DOPs implement digital object types by drawing on the notions of the OO paradigm. In order to support OO encapsulation, our approach distinguishes private from public behaviors. Private behaviors refer to operations that are executed by the digital object instance in a private fashion, hidden from third parties. For example, validations of metadata element values are private behaviors that are executed by instances according to their DOP specification, without user intervention. Private behaviors are triggered on specific events of the digital object instance's lifecycle; for instance, when a DL service updates the metadata of an object. Private behaviors are implicitly defined in the DOP, as described in the examples presented later in this section. On the other hand, public behaviors constitute the interface through which third parties can interact with the digital object instance at hand. Public behaviors are explicitly defined in a DOP and are described in Section 2.5.

2.2 Metadata Elements in DOPs

DOPs support the use of multiple metadata element sets for describing different digital object characteristics [9,10]. There are three ways to specify a metadata element set in a DOP: (a) as a standard element set, such as the Dublin Core (DC) [3], (b) as a user-defined extension of a standard element set (e.g. qualified DC) or (c) as a totally custom element set. In detail, a DOP specifies:

- the individual metadata sets contained in the objects of this type, supplied with an identifier and a multi-lingual label and description.
- the specific elements that constitute each metadata set. Each element is designated by an identifier, desired labels and descriptions, and additional behavioral characteristics expressed in terms of private behaviors.
- the possible mappings among elements of the various metadata sets.

As the `MDSets` section of Figure 2 illustrates, `Session` objects are characterized using a qualified DC metadata set, called `dc`. Due to the archival nature of the material, we also use a second, custom element set called `ead`, that follows the principles of Encoded Archival Description (EAD) [6], yet without encoding the EAD Finding Aid in its entirety.

In what follows, we describe the metadata handling capabilities of DOPs and provide appropriate examples drawn from the `MDSets` specifications found in the `Session` prototype of Figure 2.

Automatic loading & serialization of Metadata sets: Loading and serialization of metadata sets are private behaviors, both executed by the DOP behind the scenes. For example, if a DL service requests the `dc` metadata set values of a `Session` digital object instance, the DOP specified `loader` is used to load the corresponding element values from the underlying stored digital object. Respectively, whenever a DL service stores the digital object instance to the repository, the DOP supplied `serializer` is used to serialize each metadata set to the appropriate underlying format. Loaders and serializers are defined in the `datastream` XML section of the `MDSet` definition. Each DOP is allowed to define its custom loading / serialization plugins, given that they constitute valid implementations of the respective `Loader` and `Serializer` Java interfaces supplied by the *DO Dictionary*. The `Session` DOP, for example, uses the `StandardLoader` plugin to load the metadata of Session Proceedings objects.

Behavioral characteristics of Metadata elements: The DOPs metadata specification inherently offers additional behavioral characteristics for each metadata element. These characteristics are exploited by DL services on a case to case basis for each element. DOPs define behavioral characteristics in terms of XML attributes of the respective `field` definitions appearing in the `MDSet` specification. In DOPs, we support the following behavioral characteristics:

- `isMandatory`: the instance will throw an exception if the metadata element is to be saved with a null value.

- `isHidden`: advices the UI to hide the element from end-users.

- `isRepeatable`: the metadata element is allowed to have multiple values. The UI service adjusts accordingly, by supplying the cataloguer with the ability to insert multiple values or by displaying the values to the end-user in a list.

- `validation`: digital object instances apply the given validation whenever they are called to set values to the element. The validation occurs just before the user-supplied values are serialized and sent to the repository. DOPs support user-defined, pluggable validations, given that they implement the `Validation` interface provided by the *DO Dictionary*. For example, the definition of the `dc:date` element in Figure 2 specifies the use of a validation that checks whether respected values conform to the date format selected by the Senate Archive's cataloguing staff.

Mappings among Metadata Elements: The `Session` DOP of Figure 2 maps `ead:unitid` to `dc:identifier_physical`. A mapping between elements represents another example of a private behavior. Whenever the value of the `ead:unitid` element is modified, the digital object propagates its new value to the `dc:identifier_physical`. In `Session` objects, the mappings are created from selected `ead` elements to members of the `dc` metadata set. This is performed in order to allow us to offer cross-collection search to our users, given that FEDORA only supports DC metadata searches. With the use of DOP-based mappings we supply Pergamos with such search capabilities, without having to limit our material description requirements to DC metadata only or force our cataloguing staff to provide redundant information for both ead and dc metadata sets.

2.3 Digital Content in DOPs

With regard to digital content, a prototype:

- specifies the various files and their respective formats,
- provides the necessary information required for converting a primary file format to derivatives in order to automate and speed up the ingestion process,
- enables batch ingestion of content and automatic creation of the appropriate digital objects.

Listing 1.1 depicts the files configuration of the Senate Archive's Page DOP. The latter specifies that Page objects should contain three file formats, namely a high quality TIFF image (hq), a JPEG image of lower quality for web display (web) and a small JPEG thumbnail image for browsing (thumb). In what follows we describe batch ingestion and content conversion capabilities of DOPs.

```
<files>
 <file id="hq" type="primary" datastream="HQ">
  <label lang="en">High Quality Image</label>
  <mime-type id="image/tiff">
   <conversion target="web" task="convRes" hint="scale:0.6,quality:0.7"
      mimeType="image/jpeg" converter="gr.uoa.dl.core.conv.ImageConverter"/>
   <conversion target="thumb" task="convRes"
      hint="width:120,height:120,quality:0.6"
      mimeType="image/jpeg" converter="gr.uoa.dl.core.conv.ImageConverter"/>
  </mime>
 </file>
 <file id="web" type="derivative" datastream="WEB">
  <label lang="en">Web Image</label>
  <mime-type id="image/jpeg"/>
 </file>
 <file id="thumb" type="derivative" datastream="THUMB">
  <label lang="en">Thumbnail Image</label>
  <mime-type id="image/jpeg"/>
 </file>
</files>
```

Listing 1.1. The files section of the Page prototype

Automatic Digital Content Conversions: Each file format is characterized either as primary or derivative. In the case of files of Senate Archive's Page objects, as defined in the files section of Listing 1.1, the hq file is primary, referring to the original digitized material. The web and thumb files are treated as derivatives of the primary file, since the prototype's conversion behavior can generate them automatically from the hq file. Conversion details reside in the conversion section of each file specification. After the ingestion of the primary file, the digital object instance *executes the conversions residing in its prototype automatically.*

We support three conversion tasks, namely (a) `convert`, used to convert a file from one format to another, (b) `resize`, used to resize a file while maintaining its format and (c) `convRes`, used to perform both (a) and (b). Each task is carried out by the Java module supplied in the `converter` attribute, offering flexibility to users to provide their own custom converters. The converter is supplied with a `hint`, specifying either the required width and height of the resulting image in pixels, the scale factor as a number within (0, 1) or the `derivative`'s quality as a fraction of the original. In the case of `Page` objects (Listing 1.1), the `hq` file is converted to a `web` JPEG image using compression quality of 0.7 and resized using a scale factor of 0.6. Additionally, the `hq` file is also converted to a `thumb` JPEG image using compression quality 0.6 and dimensions equal to 120 x 120 pixels. The `Page` instance stores both derivatives in the FEDORA datastreams specified in the `datastream` attribute of their respective `file` XML element.

Batch Digital Object Ingestion: We also use DOPs to automate digital object ingestion. The `files` section of the `Session` prototype (Figure 2), depicts that `Session` objects are complex entities that contain no actual digital content but act as containers of `Page` objects. However, the `Session` prototype defines a `zip` file that is characterized as `container`. Containers correspond to the third supported file format. If the user uploads a file with the `application/zip` mime type in a `Session` instance, the latter initiates a `batchIngest` procedure. The `Session` DOP's `batchIngest` specification expects each file contained in the zip archive to abide to the `hq` file definitions of the `Page` prototype. In other words, if the user supplies a `Session` instance with a zip file containing TIFF images, as the `Session` `zip` file definition requires, the instance will automatically create the corresponding `Page` digital objects. Specifically, the `Session` `batchIngest` procedure extracts the zip file in a temporary location and iterates over the files it contains using the file name's sort order. If the file at hand abides to the `Page`'s `primary` file format:

a. Creates a new `Page` digital object instance.

b. Adds the `Page` instance to the current `Session` instance (as required from structural relationships described in Section 2.4).

c. Adds the file to the `Page` instance at hand. This will trigger the automatic file conversion process of the `Page` prototype, as outlined earlier.

Should we consider a `Session` comprised of 120 `Page` objects, then the ingestion automation task, supplied by DOPs, releases the user from creating 120 digital objects and making 240 file format conversions manually.

2.4 Relationships in DOPs

DOPs specify the different relationships that their instances may be allowed to participate in. Currently, DOPs support the following relationships:

- *Internal Relationships*: Digital objects reference other DL pertinent objects.
- *Structural Relationships*: These model the "parent / child" relationships generated between digital objects that act as containers and their respective "children".

- *External Relationships*: Digital object reference external entities, providing their respective URLs.

A `Session` object is allowed to contain `Page` objects; this specification appears in the `relations` section of the `Session` DOP (Figure 2). The existence of a `structure` specification in the `Session` prototype yields the following private behavior in the participating entities:

- Every `Session` object instance maintains a list of all the digital object identifiers the instance contains.
- Every `Page` instance uses the `dc:relation_isPartOf` element to hold the identifier of its parent `Session`.

Finally, the `references` part of the `relation` section informs DL services whether custom relationships are supported by this type of object. In the `Session` DOP of Figure 2, the `references` value guides UI services to allow the cataloguer to relate `Session` instances only with DL internal objects and not with external entities.

2.5 Public Behaviors in DOPs

We define public behaviors in DOPs using the notion of *behavioral scheme*. A behavioral scheme is a selection of the entities that are part of a digital object. Behavioral schemes are used to generate projections of the content of the digital object. Figure 2 illustrates the `behaviors` section of the `Session` prototype, which defines three behavioral schemes, namely `browseView`, `zipView`, and `detailView`. The `browseView` scheme supplies the user with a view of the digital object instance containing only three elements of the qualified DC metadata set, namely `dc:identifier`, `dc:title` and `dc:date`. Respectively, `zipView` generates a projection containing the `dc:title` metadata element and the `zip` file, while `detailView` provides a full-detail view of the object's metadata elements. This way, the DL designer is able to generate desired "subsets" of the encapsulated data of the digital object instance at hand for different purposes.

Execution of public behavior is performed by the invocation of a high level operation on a digital object instance, supplying the desired behavioral scheme. High level operations correspond to the actions supported by the DL modules. For example, the cataloguing module supports the `editObject`, `saveObject` and `deleteObject` actions, the browsing module supports the `browseObject` action, while object display module supports the `viewObject` action. At this stage, all Pergamos DL modules support only HTML actions:

- `viewObject("uoadl:1209", shortView)`: Dynamically generates HTML that displays the elements participating in the `shortView` of the "uoadl:1209" object in read-only mode. The *DO Dictionary* will first instantiate the object via its respective `Session` DOP (Fig. 1b). The new instance "knows" how to provide its `shortView` elements to the object display module.
- `editObject("uoadl:1209", zipView)`: Dynamically generates an HTML form that allows the user to modify the instance's elements that participate

in `zipView`. This view is used by the digitization staff in order to upload the original material and trigger the batch ingestion process, as described earlier in this section.

- `editObject("uoadl:1209", detailView)`: Generates an HTML form that displays all the metadata elements of the given instance in an editable fashion. This is used by the cataloguing staff in order to edit digital object's metadata. The cataloguing module uses the behavioral characteristics described in Section 2.2 (e.g. `isMandatory`, `isRepeatable`) to generate the appropriate, type-specific representation of the digital object.

- `saveObject("uoadl:1209", zipView)`: Saves "uoadl:1209" instance back to the repository. Only the `zipView` scheme elements are modified. Cataloguing module knows how to direct the submission of the web form generated by its aforementioned `editObject` action to `saveObject`. Respectively, cataloguing `deleteObject` action is bound to a suitable UI metaphor (e.g. a "delete" button of the web form). The scheme supplied to `deleteObject` is used to generate a "deletion confirmation view" of the digital object.

The execution of public behaviors is governed by the particular scheme at hand, while the DOP specifications enable DL application logic to adjust to the requirements of each element participating in the scheme.

3 Organization of Collections in Pergamos Using DOPs

Currently, Pergamos contains more than 50,000 digital objects originating from the Senate Archive, the Theatrical Collection, the Papyri Collection and the Folklore Collection. Table 1 provides a summary of the DOPs we generated for modeling the disparate digital object types of each collection, pinpointing the flexibility of our approach. It should be noted that DOPs are defined with a collection-pertinent scope [13] and are supplied with fully qualified identifiers, such as `folklore.page` and `senate.page`, avoiding name collisions. These identifiers apply to the object's parts, too; `folklore.page.dc` metadata set is different from the `senate.page.dc` set, both containing suitable qualifications of the DC element set for different types of objects.

a. Folklore Collection Folklore Collection consists of about 4,000 handwritten notebooks created by students of the School of Philosophy. We modeled the Folklore Collection using the `Notebook`, `Chapter` and `Page` DOPs. Notebooks are modeled as complex objects that reflect their hierarchical nature; the `Notebook` DOP allows notebooks to contain `Chapter` objects, which in turn are allowed to contain other `Chapter` objects or `Page` objects. `Notebooks` are supplied with metadata that describe the entire physical object, while `Chapter` metadata characterize the individual sections of the text. Finally, `Page` objects are not supplied with metadata but contain three files, resembling the definition of the Senate Archive's `Pages` provided in Listing 1.1.

b. Papyri Collection This collection is comprised of about 300 papyri of the Hellenic Papyrological Society. We modeled papyri using the `Papyrus` DOP, consisting of a suitable `DC` qualification and four file formats. The `orig` file format

Table 1. A summary of the DOPs we generated for four Pergamos collections

a. Folklore Collection

DOP	Metadata	Files	Relationships
`Notebook`	`dc`	none	contains `Chapter` or `Page`
`Chapter`	`dc`	none	contains `Chapter` or `Page`
`Page`	none	`hq, web, thumb`, `hq` to `web`, `hq` to `thumb` conversions	none

b. Papyri Collection

DOP	Metadata	Files	Relationships
`Papyrus`	`dc`	`orig, hq, web, thumb`, `hq` to `web`, `hq` to `thumb` conversions	none

c. Theatrical Collection

DOP	Metadata	Files	Relationships
`Album`	`custom` → `dc`	`zip` triggers batch import	contains `Photo`
`Photo`	`niso` → `dc`	`hq, web, thumb`, `hq` to `web`, `hq` to `thumb` conversions	none

d. Senate Archive's Session Proceedings

DOP	Metadata	Files	Relationships
`Session`	`ead` → `dc`	`zip` triggers batch import	contains `Page`
`Page`	none	`hq, web, thumb`, `hq` to `web`, `hq` to `thumb` conversions	none

corresponds to the original papyrus digitized image, while `hq` refers to a processed version, generated for advancing the original image's readability. The `orig` image is defined as `primary`, without conversions. The `hq` image, which is also defined as `primary`, is the one supplied with the suitable conversion specifications that generate the remaining two `derivative` formats, namely `web` and `thumb`.

c. Theatrical Collection Theatrical Collection consists of albums containing photographs taken from performances of the National Theater. Each `Photo` digital object contains three different forms of the photograph and is accompanied by the metadata required for describing the picture, either descriptive (`dc`) or technical (`niso`). As in the case of Senate Session Proceedings, mapping are used to to map `niso` elements to `dc`. `Albums` do not themselves contain any digital content, since they act as containers of `Photo` digital objects. However, `Albums` are accompanied by the required theatrical play metadata, encoded in terms of a `custom` metadata set, that is also mapped to `dc`.

d. Senate Archive The Senate Archive's Session Proceedings has been discussed in Section 2.

4 Discussion and Related Work

To our knowledge, DOPs provide the first concrete realization of digital object types and their enforcement. Our approach draws on the notions of the OO paradigm, due to its well established foundations and its well known concepts. Approaches on the formalization of OO semantics [2,12] show that the notion of objects in OO languages and the notion of digital objects in a DL system present significant similarities, yet in a different level of abstraction. [1] defines OO systems in terms of the following requirements:

- *encapsulation*: support data abstractions with an interface of named operations and hidden state,
- *type conformance*: objects should be associated to a type,
- *inheritance*: types may inherit attributes from super types.

At this stage, DOPs fulfill the encapsulation and type conformance requirements. The inclusion of inheritance is expected to provide explicit polymorphic capabilities to DOPs, since polymorphism is currently implicitly supported; the high level actions residing in the DL modules, as presented in Section 2.5, are polymorphic and can operate on a variety of types. Inheritance is also expected to allow designers to reuse digital object typing definitions. The concept of definition reuse through inheritance has been discussed in [8], although targeted on information retrieval enhancements.

Although DOPs are currently implemented atop the FEDORA repository, we believe that the presented concepts are of broader interest. The core type enforcement implementation of DOPs regarding digital object instances and their respective behavior is FEDORA independent and only stored digital object operations are tied to FEDORA specific functionality (e.g. `getDatastream`, `saveDatastream` services). Taken into consideration that DOPs, conceptually, relate to the OO paradigm and the digital object modeling approach of Kahn and Wilensky [9], we argue that there are strong indications that DOPs can be implemented in the context of other DL systems as well.

DOPs are complementary to FEDORA, or any other underlying repository. FEDORA can effectively handle low-level issues regarding digital object storage, indexing and retrieval. DOPs provide an architecture for the effective manipulation of digital objects in the higher level context of DL application logic. DOPs behaviors are divided into private and public, in order to support encapsulation, while their definition is performed in the object's respective prototype. FEDORA implements behaviors in terms of *disseminators*, which associate functionality with datastreams. FEDORA disseminators must be attached to each individual digital object upon ingestion time. With DOPs, all objects of the same type behave in the same manner; their respective behaviors are dynamically binded to the instances at runtime, while the behaviors are defined *once and in one place*, increasing management and maintenance capabilities. aDORe [4] deploys a behavior mechanism that, although it is similar to FEDORA, it attaches behaviors to stored digital objects in a more dynamic fashion, upon dissemination time, using disseminator-related rules stored in a knowledge base. Finally, DOPs behaviors operate on digital objects in a more fine-grained manner, since they can explicitly identify and operate upon the contents of FEDORA datastreams.

[5] enables the introspection of digital object structure and behavior. A DOP can be conceived as a meta-level entity that provides structural and behavioral metadata for a specific subset of base-level digital objects. Put in other terms,

a DOP acts as an introspection guide for its respective digital object instances. DOP supplied type conformance and type-driven introspection of digital object structure and behavior allows third parties to adjust to each object's "idiosyncrasy" in a uniform manner.

References

1. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
2. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Application (OOPSLA)*, pages 433–444, New Orleans, Louisiana, USA, 1989.
3. *DCMI Metadata Terms.* Dublin Core Metadata Initiative, January 2005.
4. H. Van de Sompel, J. Bekaert, X. Liu, L. Balakireva, and T. Schwander. adore: A modular, standards-based digital object repository. *The Computer Journal*, 48(5):514–535, 2005.
5. N. Dushay. Localizing experience of digital content via structural metadata. In *Proceedings of the Joint Conference on Digital Libraries*, pages 244–252, Portland, Oregon, USA, 2002.
6. *Encoded Archival Description (EAD).* Library of Congress, 2006.
7. *Introduction to Fedora Object XML.* Fedora Project.
8. N. Fuhr. Object-oriented and database concepts for the design of networked information retrieval systems. In *Proceedings of the 5th international conference on Information and knowledge management*, pages 164–172, Rockville, Maryland, USA, 1996.
9. R. Kahn and R. Wilensky. *A Framework for Distributed Digital Object Services.* Corporation of National Research Initiative - Reston, VA, 1995.
10. *METS: An Overview & Tutorial.* Library of Congress, Washington, D.C., 2006.
11. *Resource Description Framework (RDF).* World Wide Web Consortium.
12. U.S Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 289–297, Snowbird, Utah, USA, 1988.
13. K. Saidis, G. Pyrounakis, and M. Nikolaidou. On the effective manipulation of digital objects: A prototype-based instantiation approach. In *Proceedings of the 9th European Conference on Digital Libraries*, pages 26–37, Vienna, Austria, 2005.
14. T. Staples, R. Wayland, and S. Payette. The fedora project: An open-source digital object repository management system. *D-Lib Magazine*, 9(4), April 2003.
15. T. Staples, R. Wayland, and S. Payette. Using mpeg-21 dip and niso openurl for the dynamic dissemination of complex digital objects in the los alamos national laboratory digital library. *D-Lib Magazine*, 10(2), February 2004.