

# Controlled Migration in Digital Archives

Thomas Triebsees, Uwe M. Borghoff, and Jan Scheffczyk

Universität der Bundeswehr München

{Thomas.Triebsees, Uwe.Borghoff, Jan.Scheffczyk}@unibw.de

**Abstract.** Since quite a while, long term preservation of digital information has become a challenging issue. Migration and emulation are two approaches that have been suggested for making today's knowledge available for future generations. Although migration, in particular, is progressively being used in digital archiving, there is still a lack of formalization. In planning and running comprehensive migration processes, librarians and archivists face the major challenge of preserving the original information. To reduce unrecognized information change or information loss, it would be useful to have a formal framework allowing to *qualitatively* specify information. Quantitative and probabilistic approaches lack the possibility to discover *when which* information changes or is lost.

In this paper, we propose how to capture the notion of information and to qualify information loss during migration processes. Furthermore, we use the Unified Modeling Language (UML) to exemplary design an information structure and illustrate some major problems that arise when a migration is planned and executed.

## 1 Introduction and Motivation

Over the past decade, long term preservation of digital heritage has become an urgent issue. The main reason is the exponential growth of digital material which originates from the rapid deployment of personal computers and the evolution of the world wide web. By “long term” we mean a period of time that is long enough to be potentially affected by significant technology changes [1]. Therefore, digital archives face the challenge to keep electronic data accessible, viewable and usable. Migration beyond others has emerged to be a promising strategy for that purpose [2]. The Reference Model for Open Archival Information Systems (OAIS, [1]) identifies four aspects of migration: Replication, refreshment, repackaging and transformation. The first three are well-handled in practice. We concentrate on the latter and henceforth identify migration with transformation.

Two major concerns of migration are 1) to transform digital objects into formats that are used by the archive and have a good chance to be supported for a long period of time and 2) to reorganize the storage structure of the preserved objects in order to face changing general conditions. The overall aim is to minimize unintended information loss during the transformation process and to preserve object relationships. Although migration is widely practiced, we are not aware of comprehensive, theoretically well-founded methods to support digital

archives in planning and executing migrations. Even state-of-the-art tools lack well-founded support in this respect [3]. Hence, object migration is still an error prone task and often relies on brute-force and heuristic methods.

The key challenge is that digital information objects can have very complex structures, including several kinds of meta data (e.g., representation information). Today, complex meta data sets are often connected to information objects in order to describe their context and to allow for semantic indexing and searching. The overall goal is to make the correct interpretation possible for future generations. The important role of object context was also emphasized in [4] where ontologies and semantic annotation graphs are used to create meaningful contexts. Migration must preserve context information.

Proprietary or domain-specific formats are also problematic. Proprietary file formats often require expensive commercial transformation tools. Structure elements of domain specific formats might not be transformable from one format to another. Hence, open standards like XML or de-facto standards like PDF are the most promising file formats for long term preservation. XML as self-describing format opens a wide range of possibilities.

In this paper we investigate how to formalize information and how to manage object migration in the context of digital archives. We propose an appropriate information model allowing to qualify information. Moreover, we integrate this model into current technologies by using the Unified Modeling Language (UML, [5]) to visualize information. UML, thus, serves as a bridge between the necessary formalism and user acceptance. More importantly, there is a variety of tools for UML. The main contribution of this paper is that we provide a coherent view on information in a digital archive and its representation using UML. The suitability of our approach is illustrated. We exemplarily show that it is flexible enough to face the particularities of digital archiving. Complex object dependencies are identified and their preservation during a migration process is examined.

The road map for the rest of the paper is as follows: First, we introduce a running example. In Section 2 we describe the information model, which is the basis to qualify information and information loss during migration. We use UML to model information and complex object dependencies. In Section 3 we analyze an example migration and point out some problems. Related work is described in Section 4. In Section 5 we conclude with a summary and a brief outlook.

## 1.1 Running Example

We consider migration within a digital archive for newspapers. Since instances of one article might occur in different newspapers, articles and newspapers are stored separately. To support efficient full text search and indexing, both are stored as text files and encoded in an XML format. Figure 1 exemplarily shows the DTD `Newspaper.dtd` for newspapers.

Articles consist of a globally unique ID, a headline, and a content. Pictures used in the article are stored separately and referenced from within the content. Newspapers also have an ID. Additionally, they have a header consisting of the name of the newspaper and its publishing date. The content is composed of

Newspaper.dtd	NewsDescr.dtd
<pre> &lt;!ELEMENT Newspaper (Header,Content,DigImage)&gt; &lt;!--ATTLIST Newspaper id ID #REQUIRED--&gt; &lt;!ELEMENT Header EMPTY&gt; &lt;!--ATTLIST Header name CDATA #REQUIRED date CDATA #REQUIRED --&gt; &lt;!ELEMENT Link EMPTY&gt; &lt;!--ATTLIST Link page CDATA #REQUIRED xmlns:xlink CDATA "http://..." xlink:type ... #FIXED "simple" ... xlink:actuate ...--&gt; &lt;!ELEMENT Content (Link*)&gt; &lt;!ELEMENT DigImage (Link+)&gt; day CDATA #REQUIRED </pre>	<pre> &lt;!ELEMENT NewsDescr (DateViewed,Location, LocDigImage) &lt;!--ENTITY % linkPars "xmlns:xlink ...&gt; xlink:type ... ... xlink:actuate ...--&gt; &lt;!--ATTLIST NewsDescr id ID #REQUIRED newsID IDREF #REQUIRED %linkPars; --&gt; &lt;!ELEMENT Location #PCDATA&gt; &lt;!--ATTLIST Location locID IDREF #REQUIRED--&gt; &lt;!ELEMENT DateViewed EMPTY&gt; &lt;!--ATTLIST DateViewed year CDATA #REQUIRED month CDATA #REQUIRED --&gt; &lt;!ELEMENT LocDigImage (Location+)&gt; </pre>

Fig. 1. DTDs for newspapers and their meta data objects

article references. To have access to the original newspapers, a digital image is stored. We assume that one image file is available for each page of the respective newspaper. This is indicated by the attribute `page` of the element `Link`. Furthermore, all image files shall be stored using the TIFF format.

Consequently, the textual content of newspapers can be reproduced by the referenced articles. In contrast, references to digital images give access to the original document. Note that we have two kinds of picture references. In articles, links point to pictures used as illustration. A reference in a newspaper points to the digital image of one particular page of that newspaper.

Additional meta data for articles and newspapers is stored. We again use XML DTDs to describe their structure (cf. Figure 1). Each article is accompanied by a meta data object describing its storing location, author, and occurrences. The set of newspapers, in which an article occurs, is computed using the references to that article from within the newspapers. The meta data objects for newspapers contain the storing location of the respective newspaper, the locations of its image files and the date on which it was last viewed.

**Migration Purpose.** We assume a change request for our example archive. The new preservation strategy results in newspaper objects that directly include the related digital image files. Furthermore, article objects shall include their pictures. This strategy of storing *records* reduces link-consistency and access problems. A full implementation of this strategy can be found in [6]. The separation of articles and newspapers, however, shall be maintained to have better access to single articles. The stored records shall again be XML files. To include the binary data from the attached pictures, we use base64 encoding [7].

Both, article and newspaper records, need additional meta data. We maintain an extra object for each article and newspaper record to locate them.

We will see that even this simple XML-based example suffices to show major information preservation problems. The problems would get even worse and more complex if binary file formats were used.

## 2 Information Model

An appropriate information model is the basis for an exact, but still practicable, treatment of migration. We do not aim at probabilistic predictions about information lost during a migration process. We rather want to qualify *what* information is lost and *when* it is lost.

A look at the running example shows that information contained in an object can be described by attributes and functions defined on the object (for example *getID()* to get the ID of a newspaper). Concerning the internal object structure, we can define invariants on its value. Newspapers, for example, must be valid XML documents according to `Newspaper.dtd`.

In the following we will give an overview of this approach and examine it using our running example. We employ a formalism from the context of object oriented programming [8]. The key point is to develop a type system and uniquely assign a type to each information object. A type specification then provides an interface to information retrieval methods for objects of that specific type.

### 2.1 The Type System

We formalize a type  $\tau$  as a tuple  $(I, O, V, M)$  where

1.  $I$  is an invariant which must hold for all values of type  $\tau$ ,
2.  $O$  is the set of objects of type  $\tau$ ,
3.  $V$  is the set of values that may be assigned to objects of type  $\tau$ ,
4.  $M$  is the set of methods defined for  $\tau$ .

The tuple's components are called *inv*, *obj*, *val* and *meth* henceforth. Thus,  $obj(\tau)$  describes the objects of type  $\tau$ . We abbreviate  $o \in obj(\tau)$  by  $o : \tau$ . There is an important difference between objects and values. Objects are referable units that have a value uniquely assigned to them. Consequently, we speak of *the* value of an object. A file, for example, is an object because it is referable. The value of the file is its content. From now on, we denote the value of an object  $o$  by  $value(o)$  and the type of  $o$  by  $type(o)$ .

Every method specification consists of a name, a signature, a pre condition, and a post condition. We will denote the parts of a method specification for a method  $m$  by  $name(m)$ ,  $sig(m)$ ,  $pre(m)$  and  $post(m)$ . The  $i$ th argument of a method  $m$  is denoted by  $arg(i, m)$ . Figure 2 illustrates the type *Article* as UML class. We use UML since it is a standardized, well-accepted modeling language that provides all necessary modeling elements relevant to our context. In particular, complex object and type dependencies can be modeled easily. We use natural language to express class invariants and a more formal one (similar to OCL) for pre and post conditions.

In Figure 2 we have one private attribute for each piece of information, namely *id*, *headline*, *content*, *refs*. Two constructor methods with different typing are provided. Public getters (*getID()*, *getHeader()*, etc.) give access to the attributes. The methods' pre and post conditions are shown in braces following

<b>Newspaper</b>	
{ Valid XML file according to <b>Newspaper.dtd</b> }	
- id : ID - header : String - artRefs : List - imRefs : Set	
<<create>>Newspaper():Newspaper <<create>> Newspaper(id:ID, h:String, arts:List, imRefs:Set):Newspaper	{ <<pre>> id.isValid(), <<post>> self.id=id and header=h and artRefs=arts and self.imRefs=imRefs }
+ getID():ID	{ <<pre>> True, <<post>> result = id }
+ getArtList():List	{ <<pre>> True, <<post>> result = artRefs }
+ getHeader():String	{ <<pre>> True, <<post>> result = header }
+ getImageRefs():Set	{ <<pre>> True, <<post>> result = imRefs }
+ toString():String	{ <<pre>> True, <<post>> result = id.toString() + " " + header + " " + arts.toString() }

**Fig. 2.** The type *Newspaper* as UML class

the respective signature specification. The pre condition for the second constructor, for example, specifies that the given ID has to be valid. The *isValid()* method must be defined for the type *ID*. The post condition asserts that the private attributes are set properly. Since pre and post conditions are specified formally, they allow for formal validation and proofs.

The type description for an article is part of a whole type system. For example, we also have to provide a type *ID*. We denote a type system by  $\mathcal{T}$ .

There is one important aspect we want to emphasize. First, we require all methods of all types to be referentially transparent, i.e., computed results depend on the arguments only. In this way we distinguish between context knowledge and information carried within an information object. The context of an object can change which may induce a migration of the object. The migration obligation, however, can only be determined if we can decide whether the context change influences the information carried by the object. To do so, we must be able to qualify context information. Hence, object context also has to be stored within information objects (meta data objects). New context knowledge can be brought into the system by meta functions, as we will see in Section 3. Consequently, we prohibit static methods. In this respect, the class design of Figure 2 is significant for all types. The *private* attributes are set once upon creation of the object. Their value can be accessed exclusively by getters. Subsequent changes are impossible. No static methods are provided.

**Subtyping.** Type systems can grow very large. Suppose, for example, that each structure element of a large XML schema shall be represented by an own type. At this, a re-use mechanism can provide a big benefit. Subtyping structures the type system by introducing an inheritance mechanism. The notion “is subtype of” then corresponds to an *is-a* relation.

Let  $\tau := (I, O, V, M)$  and  $\sigma := (I', O', V', M')$  be two types from the same type system. Then  $\sigma$  is a subtype of  $\tau$  (denoted by  $\sigma \leq \tau$ ) if there is an ab-

straction function  $a : val(\sigma) \rightarrow val(\tau)$  and a renaming function  $r : meth(\sigma) \rightarrow meth(\tau)$  such that the following holds:

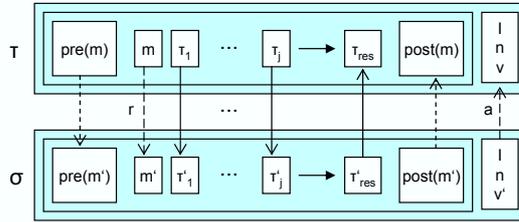
1.  $I'(v')$  implies  $I(a(v'))$  for all values  $v' \in V'$ .
2. For all methods  $m$  of  $\sigma$  and their counterpart  $r(m)$  of  $\tau$  the following conditions hold:
  - $m$  and  $r(m)$  have the same number  $i$  of arguments and  $arg(j, r(m)) \leq arg(j, m)$  for all  $j \in \{1, \dots, i\}$ .
  - The result type of  $m$  is a subtype of the result type of  $r(m)$ .
  - $pre(r(m))$  implies  $pre(m)$ .
  - $post(m)$  implies  $post(r(m))$ .

Having types  $\sigma$  and  $\tau$  with  $\sigma \leq \tau$ , we would like to use objects of  $\sigma$  as if they were of type  $\tau$  without recognizing any differences in the behavior w.r.t. the functionality of  $\tau$  (Liskov's substitution principle [8]). We achieve this by demanding contra variance in the argument types of  $m$  and  $r(m)$  and covariance in the result types. If we call a method  $m \in meth(\tau)$  on an object of type  $\sigma$ , these rules ensure that this call induces a valid call of the counterpart of  $m$  in  $\sigma$  (contra variance of arguments) and the delivered result is also a valid result of  $m$  (covariance of results). The same is true for the pre and post conditions. In Figure 3 these interactions are demonstrated for types  $\sigma \leq \tau$ , both having one method. Different kinds of arrows represent different kinds of interactions.

Let us, for example, introduce a most general type  $Obj$  providing the function  $toString()$  (Figure 4). We immediately conclude that  $Newspaper$  is a subtype of  $Obj$  with identity as abstraction (i.e.  $a(v) := v$  for all values  $v$  of type  $Newspaper$ ) and renaming  $r$  given by  $r(toString) := toString, r(Newspaper) := Obj$ . The invariant of  $Newspaper$  implies the invariant of  $Obj$  since the implication  $\Rightarrow True$  always holds. Moreover, the method and pre and post condition rules, respectively, are satisfied under renaming. The invariant  $True$  will be omitted henceforth.

The conditions introduced above are insufficient for mutable objects [8]. For our purposes, however, immutable objects suffice. Changing the content of objects is a transformation and, hence, necessitates the creation of a new object.

As usual, the subtype relation is reflexive and transitive. Antisymmetry is not required, i.e., we allow equivalent types. Since we require a most general type  $Obj$ , a type system  $\mathcal{T}$  together with a subtype relation  $\leq$  is a preorder with



**Fig. 3.** Subtyping mechanism

greatest element  $Obj$ . Note that the type system can be regarded as a directed graph where the nodes are the types and the edges reflect the subtype relation. The type graph corresponding to  $\mathcal{T}$  will be denoted by  $G_{\mathcal{T}}$ . Object dependencies can be modeled by labeled, weighted hyper edges. The weight of a specific edge is a relation connecting all dependent objects of the respective types. Adding these edges to  $G_{\mathcal{T}}$  gives a coherent view on type and object dependencies.

**Comparing Objects.** As we aim at a model for object migration, we need a mechanism to compare objects before and after transformation processes. Only this facilitates qualification of information change or loss.

Two objects are called equal if and only if they have exactly the same values and the same type, i.e.

$$o_1 = o_2 \quad :\Leftrightarrow \quad value(o_1) = value(o_2) \text{ and } type(o_1) = type(o_2).$$

In contrast, two objects are *undistinguishable* by the methods of type  $\tau$  if and only if applying the methods of  $\tau$  to  $o_1$  and  $o_2$  with any possible arguments delivers the same result. Mathematically spoken, this means

$$o_1 \approx_{\tau} o_2 \quad :\Leftrightarrow \quad \forall m \in meth(\tau) \bullet \\ \forall o'_1 : arg(1, m), \dots, o'_i : arg(i, m) \bullet \\ ( a_1(o_1).m(o'_1, \dots, o'_i) = a_2(o_2).m(o'_1, \dots, o'_i) )$$

where  $a_1$  and  $a_2$ , respectively, denote the abstraction functions from  $type(o_1)$  and  $type(o_2)$  to  $\tau$ , respectively. A direct consequence is that  $type(o_1)$  and  $type(o_2)$  have to be subtypes of  $\tau$  whenever  $o_1 \approx_{\tau} o_2$  holds.

Furthermore, equality of two objects implies that they are undistinguishable by the methods of their type. This conclusion is not trivial since it follows from referential transparency of all methods.

## 2.2 The Full Example Type System

The full type system for the running example is shown in Figure 5. Recall the class style of Figure 2. It is also used here whereas attributes are omitted for brevity. The same is true for class details that have already been introduced above. Furthermore, trivial pre and post conditions for constructors and methods, and trivial constructors are not shown or abbreviated.

The provided functionality directly reflects the description of Section 1.1. In particular, we introduce two record types *NewsRec* and *ArticleRec* (not shown

<b>Obj</b>	
{ True }	
<<create>> Obj() : Obj	{ <<pre>> True, <<post>> True }
+ toString():String	{ <<pre>> True, <<post>> True }

**Fig. 4.** Most general type *Obj*

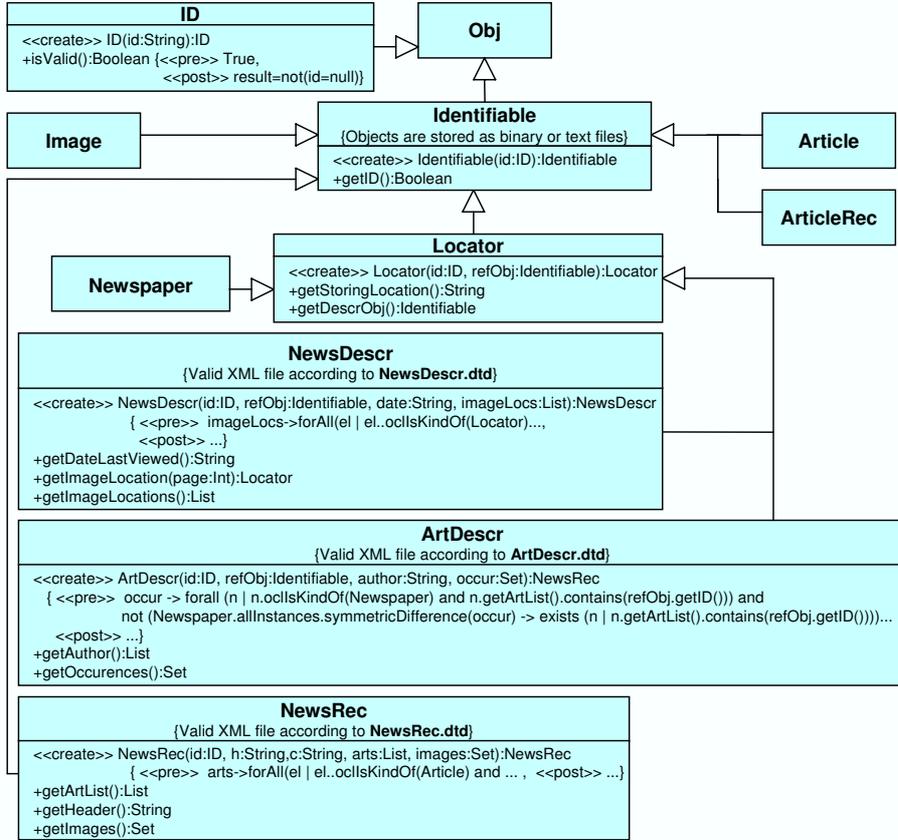


Fig. 5. The full type system for the running example

in detail). The methods of type *NewsRec* coincide with our purpose to integrate newspapers and their related image files into records.

Some pre and post conditions are of particular interest. The class *ArtDescr* shows how to specify context knowledge and still respect referential transparency. The pre condition of the second constructor quantifies over all instances of class *Newspaper*. This potentially violates referential transparency since number and extent of these instances depend on time. This side effect, however, is necessary to express that the parameter *occur* must be a set of exactly those *Newspaper* objects that contain a reference to the described article. By using the pre condition of the constructor, we specify that the required context knowledge must already be provided upon creation of the object. In contrast, the specification in question would disobey referential transparency if placed in the post condition of *getOccurrences()*. The remaining parts of this diagram are straightforward.

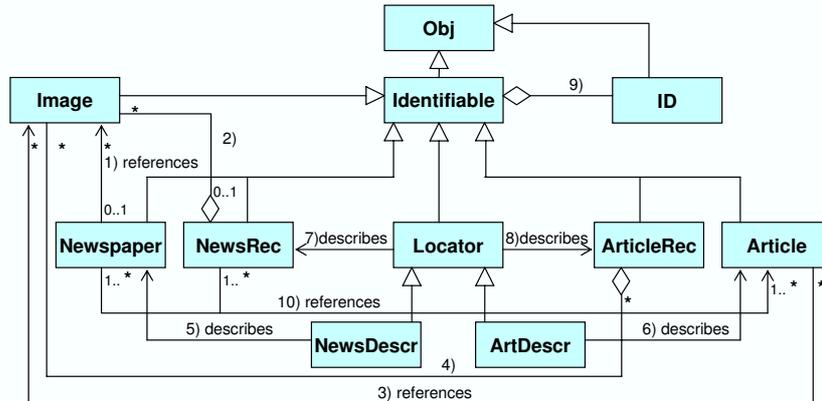


Fig. 6. UML class diagram showing major dependencies

### 2.3 Discussion of the Information Model

The information model proposed here has several advantages. The object oriented approach directly reflects the understanding of preserving *objects*. Supplemented by UML, it guarantees tool support and easy visualization, which is important for user acceptance. This is indispensable since librarians and archivists usually are no computer scientists or mathematicians. UML offers a variety of standard constructs and the possibility to incorporate user-defined ones. Hence, dependencies on the object or type level can be incorporated and modeled in a natural way. This, for instance, supports definition of ontologies.

Our approach also allows for an exact mathematical treatment. Formally specified types and subtype relations and referentially transparent methods are the basis to precisely reason about objects, their dependencies, information change, and information loss. Thereby, all degrees of granularity are supported.

## 3 Migration Shown Exemplary

In this section we study the sample migration task outlined in Section 1.1. We identify type and object dependencies, show how to preserve them during the migration, and sketch how they may conflict. Moreover, we outline required parameters for the migration function and derive an algorithm to run the migration.

### 3.1 Incorporating Dependencies

Figure 5 shows subtype relations only. But our running example indicates additional dependencies which are shown in Figure 6.

Dependencies are numbered for ease of reference. *Newspaper* and *Article* objects contain image references (1, 3). The main difference is that newspapers refer to their digital images and articles refer to included pictures. In contrast

to normal pictures in articles, digital images of newspapers are unique. This is reflected by the multiplicities. The respective record objects *NewsRec* and *ArticleRec* have similar dependencies (2, 4). Image objects shall be incorporated into record objects, which corresponds to an aggregation according to UML terminology. The multiplicities are similar. Moreover, we have meta data relations expressing that each *Article* and *Newspaper* object is described by exactly one *ArtDescr* and *NewsDescr* object, respectively (5, 6). Analogously, locators describe newspaper and article records (7, 8). An *ID* is part of type *Identifiable* (9). Finally, *Newspaper* and *NewsRec* objects refer to articles (10).

There is a more subtle dependency between the function *getOccurrences()* (*ArtDescr*) and the function *getArtList()* (*Newspaper*). The former returns those newspapers in which the respective article occurs (by returning the parameter *occur* of the constructor). But the precondition of the constructor uses *getArtList()* to specify the parameter *occur*. This considerably influences the migration process as we will see later on. In Figure 6 we have not explicitly shown such *functional dependencies* for clarity.

We can distinguish two classes of dependencies: Type level dependencies and object level dependencies. The former relate types disregarding instances of the related types. Subtype relations, ontologies, and functional dependencies, for example, constitute type level dependencies. In contrast, object dependencies are  $m : n$  relationships between objects of the respective types. Except the subtype relation, all relations shown in Figure 6 embody object level dependencies.

### 3.2 Preserving Dependencies

In Section 1.1, the migration purpose was described relatively vague. Therefore, we state the transformation objectives more precisely as follows:

1. A single *Newspaper* object and all its related *Image* objects shall be transformed into a single *NewsRec* object.
2. A single *Article* object and all its related *Image* objects shall be transformed into a single *ArtRec* object.

The two record types were introduced in very similar ways and, hence, can be handled almost identically. In the following we consider transformation 1 and denote the transformation function by  $\delta$ . According to the transformation objective,  $\delta$  is of type  $Newspaper \times Set(Image) \rightarrow NewsRec$ . The first objective, however, says that newspapers and their *related* images shall be transformed. This shows that  $\delta$ , in general, is a partial function.

After the migration process all *Newspaper* and their *related Image* objects shall be deleted. The other option would have been to maintain them. Since the image files are included in the new record objects, we avoid having them stored double and prefer the first option.

In order to include the full dependency knowledge into the migration  $\delta$ , we have to traverse all dependencies related to the source types of  $\delta$ . Dependencies can cause additional *migration obligations*. The intuitive understanding of subtyping, for example, is: Doing something with all vehicles means doing it with

all cars, motorbikes, and so on. In our example, *Newspaper* has no subtypes so that we do not need to consider any additional types in this respect. There is, however, a relation between *Newspaper* objects and articles (10). The same relation holds for *NewsRec* objects. Since  $\delta$  shall not change any article references, we automatically preserve this dependency. In contrast, the meta data dependency between newspapers and their description (3) cannot be maintained that easily because we have no appropriate dependency between *NewsDescr* and *NewsRec*. Hence, we must also migrate type *NewsDescr*. This is done by an additional transformation function  $\delta_2 : \text{NewsDescr} \rightarrow \tau$ . Since  $\delta_2$  is a migration obligation, we must preserve the dependency structure concerning *Newspaper* and *NewsDescr* objects. In Figure 6 we see that each *NewsRec* object must be described by exactly one *Locator* object (7). Hence, we must migrate type *NewsDescr* to *Locator* (i.e.,  $\delta_2$  has typing  $\delta_2 : \text{NewsDescr} \rightarrow \text{Locator}$ ).

Images might also be referenced by articles or included in article records. But we require that images referenced by articles and newspapers, respectively, are distinct (because of their different purpose). Formally, this means that the relations constituting the weights of edge 1) and 3) in  $G_{\mathcal{T}}$  have different targets. Therefore, we can ignore this fact.

Finally, we must take into account a hidden dependency that might be overseen. We already mentioned that the function *getOccurrences()* of type *ArtDescr* is functionally dependent on the function *getArtList()* of type *Newspaper*. Our migration purpose is to migrate all *Newspaper* objects to type *NewsRec* and then delete them. Consequently, we must decide whether *Newspaper* shall remain a valid type. Since we made the strategic decision to switch to newspaper records, we want the type *Newspaper* to be invalid once the migration process has been completed. Therefore, type *ArtDescr* also has to be transformed. Since the method *getArtList()* is also available for *NewsRec* objects, it suffices to set up a type *ArtDescr'* by changing all occurrences of *Newspaper* to *NewsRec* in the type specification of *ArtDescr*. The transformation then is done by a function  $\delta_3 : \text{ArtDescr} \rightarrow \text{ArtDescr}'$ . Since *ArtDescr* has no incoming dependencies, there are no further obligations.

The most interesting point, however, is to qualify the information preserved by the transformation functions. First, we examine  $\delta$ . Since base64 encoding is a bijective mapping, we preserve type *Set(Image)* (indicated by the method *getImages()* of type *NewsRec*). Moreover, type *Newspaper* is maintained except the method *getImageRefs()*, its related attribute *imRefs*, and the second constructor. Denoting the induced type by *Newspaper'*,  $\delta$  preserves *Set(Image)*  $\times$  *Newspaper'* on the whole. Recalling Section 2.1, this is expressed by

$$\forall o : \tau \bullet o \approx_{\tau'} \delta(o)$$

where  $\tau := \text{Newspaper} \times \text{Set(Image)}$  and  $\tau' := \text{Newspaper}' \times \text{Set(Image)}$ .  $\delta_2$  affects *NewsDescr* objects. The methods *getDateLastViewed()*, *getImageLocation()* and *getImageLocations()* are lost and the information delivered by the *Locator* methods *getStoringLocation()* and *getDescrObj()* is changed. Hence, we have  $\forall o : \text{Locator} \bullet o \approx_{\text{Identifiable}} \delta_2(o)$ . A transformation reflecting the

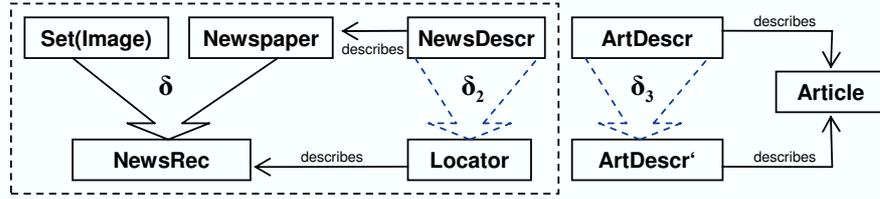


Fig. 7. The full transformation process

main migration purpose together with the induced transformations is understood as migration. Their conjunction is essential to obtain transformation results that respect existing dependencies. Figure 7 shows the whole transformation process. The transformations  $\delta_2$  and  $\delta_3$  are marked by a dashed line to express that they are induced.  $\delta$  and  $\delta_2$  are put into one box since they are strongly coupled according to the meta data relationship.

### 3.3 The Semantics of the Migration Process

Now, we specify the semantics of the whole migration. Recall that  $\delta$  reflects the main migration purpose. The migration process can be described as follows.

1. Take a *Newspaper* object, its related set of images and the describing *NewsDescr* object.
2. Use  $\delta$  to transform the *Newspaper* object and the related images.
3. Use  $\delta_2$  to transform the *NewsDescr* object using the result of  $\delta$ .
4. If there are still *Newspaper* objects left, go to 1.
5. Take an *ArtDescr* object and transform it using  $\delta_3$  and the transformation results of  $\delta$ .
6. If there are still *ArtDescr* objects left, go to 5.
7. Delete all objects that have been transformed by  $\delta$ ,  $\delta_2$ , or  $\delta_3$ .
8. Mark type *Newspaper* as invalid.

We directly observe that  $\delta$  and  $\delta_2$  are dependent in the way that the affected source objects of  $\delta_2$  depend on the source objects of  $\delta$ . This is not true for  $\delta_3$ . Essentially,  $\delta_2$  has to “wait” for the result of  $\delta$ . The reason is that the resulting *Locator* object must be instantiated with the ID of the resulting *NewsRec* object of  $\delta$ . Consequently, cyclical dependencies must be resolved or even forbidden in order to avoid dead locks.  $\delta_3$  can only be executed after all newspapers are migrated. To transform *ArtDescr* objects, all *NewsRec* objects must be known to provide the parameter *occur* correctly. Again, dead locks have to be avoided.

The carefully provided execution order of  $\delta$ ,  $\delta_2$ , and  $\delta_3$  shows that it is a non-trivial task to formally specify migrations and their semantics.

### 3.4 Lessons Learnt

The last sections have brought up a variety of relevant tasks.

1. A type system must be specified.
2. Object dependencies have to be discovered.
3. The migration objectives must be formulated.
4. Induced migration obligations are to be identified.
5. The preserved information must be qualified.
6. The migration has to be executed in a way allowing to preserve object dependencies.

The specification of a type system is critical because unspecified information can cause considerable problems. The translation of large XML schemas into UML class diagrams, for example, can hardly be managed manually. Hence, tool support is crucial. But many state-of-the-art UML tools support reverse engineering for different languages including XML and XML schema <sup>1</sup>.

Object dependencies can be generated automatically if the design principles of Figure 2 are respected. The second constructor of type *ArtDescr*, for example, gets an object of type *Article*. This facilitates the deduction of a relationship between articles and their meta data objects. By stereotypes we indicate type and multiplicities of relationships. Moreover, we automatically generate functional dependencies from the pre and post conditions. Doing so, we profit from the formal information model. Again, tool support is important. An OCL and UML validation tool, for example, is described in [9]. Moreover, there are many other tools available supporting formal specification, validation, and reasoning.

Formal specifications of migration objectives together with our information model ensure induced migration obligations to be computed in a fully automated way. The preserved information has to be specified for the main transformation, but can be deduced automatically for the induced transformations. The execution semantics can be deduced provided the migration objective is specified in sufficient detail. This avoids conflicts arising from wrong execution orders.

To sum up, the formal information model is our basis for a comprehensive, formal approach to capture migration processes in digital archives. Together with the representation and visualization offered through UML, we gain user acceptance and tool support. The latter allows for comprehensive automation making planning and execution of migration tasks less error-prone.

## 4 Related work

There are some approaches to formalize migration in other contexts. Process migration is, for instance, dealt with in [10]. There, process transfer, or better process migration, serves as the basis for improving the workload among interconnected workstations. The major problems are to realize the migration need, to set up a time schedule, and to develop a strategy to coordinate process transfers. The models are of stochastic nature and cover a different domain. The notion of information objects and the distinction between type level and object level dependencies are not captured.

---

<sup>1</sup> E.g., Softpedia's Visual Paradigm for UML

Database migration is another approach [11]. The aim is to transform one database schema into another while preserving as much information as possible. Reversible transformations are of major interest since usually the data is not transformed physically. In fact, queries are translated from the new schema to the old one. Therefore, this approach mainly focuses on the schema transformation function itself. We see our migrations as a black box only knowing the preserved type of information. This allows for considerations concerning interactions of migration functions on a higher level.

[12] describes ontology merging as a migration process. It aims at merging two ontologies in a way where both are semantically joined to a new ontology. The source ontologies are possibly described in different languages. This might necessitate to translate them into a “meta” language. After that, the least ontology including both source ontologies has to be computed. In analogy to database migration, this approach concentrates on the translation of the source languages into the destination language.

The Typed Object Model (TOM, [13]) incorporates and deploys type descriptions via a type brokerage system. The main focus lies on transformations between diverse file formats and qualification of information loss. We borrow some ideas from TOM, especially the type system. TOM, however, lacks a clear separation of contextual knowledge. As a consequence thereof, methods need not be referentially transparent. This is manifested by transformation functions being defined on types rather than on a meta level. We, in contrast, consider transformations to be meta functions. They are the only possibility to incorporate contextual knowledge into the system. In this way we ensure all relevant knowledge to be stored in information objects. Moreover, TOM offers no mechanism to support comprehensive migration strategies that respect object dependencies. As shown in Section 3, this requires considerable extra effort.

## 5 Conclusion and Outlook

In this paper we have focused on planning and executing migration tasks in digital archives. We have proposed a formal information model capturing the notion of information and information change/loss during a migration process. We have outlined the advantages of using UML to represent and visualize information in this context. Employing an example migration task, we have identified complex object and type level dependencies, and have shown their impact on the migration plan. Finally, we gave a sketch of the major problems and how they can be solved in a single comprehensive, formal framework. We have identified that tool support and automation are crucial points. Especially, the latter requires a sufficiently formal approach.

Future work will cover three major issues. First, we will fix the design principle for UML classes to ensure a coherent type design and, hence, facilitate automated deduction of object and type dependencies. We plan to use the Meta Object Facility (MOF) to develop a meta meta model for the class design.

Second, we plan to provide for a formalism describing migrations. We already gave a sketch of some important parameters in Section 3. This is a crucial task because a sufficiently formal description is indispensable in order to test migrations for consistency and feasibility. Conflicts between object dependencies and the migration purpose, for example, are then detected in a fully automated way. Since migration tasks can be rather complex, users are overwhelmed when managing them all manually.

Finally, we must specify a formal semantics for migrations. This ensures migrations to be understood w.r.t. their effect on the archive and their impact to information objects. Integrating these three aspects into the information model can close the gap between ad hoc migration on the one hand, and sound, theoretically founded, tool supported migration on the other hand.

## References

1. Consultative Committee for Space Data Systems CCSDS: Reference model for an open archival information system (OAIS). Technical report, Space Data Systems (2002)
2. Borghoff, U.M., Rödiger, P., Scheffczyk, J., Schmitz, L.: Langzeitarchivierung. dpunkt Verlag, Heidelberg (2003)
3. Borghoff, U.M. et. al.: Vergleich bestehender Archivierungssysteme. nestor c/o Die Deutsche Bibliothek, Frankfurt am Main (2005)
4. Kitamoto, A., Sato, S., Yamamoto, T., Ono, K.: Context recombination for digital cultural archives. In: Proc. Int. Conf. on Digital Archive Technologies (IC-DAT2004). (2004) 105–119
5. The Unified Modeling Language (UML), <http://www.uml.org>.
6. VERS: Victorian electronic records strategy - final report. Technical report, Public Record Office Victoria (1998)
7. Internet Engineering Task Force (IETF): RFC 1521 - MIME Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. (1993)
8. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems **16** (1994) 1811–1841
9. Gogolla, M., Richters, M., Bohling, J.: Tool Support for Validating UML and OCL Models through Automatic Snapshot Generation. In: Proc. Annual Research Conf. South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT'2003). (2003) 248–257
10. Milošević, D., Douglass, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. Technical report, Computer Systems Laboratory, HP Labs Palo Alto (1999)
11. McBrien, P., Poulouvasilis, A.: A uniform approach to inter-model transformations. In: Proc. 11th Int. Conf. on Advanced Information Systems Engineering (CAiSE'99). (1999) 333–348
12. Stumme, G., Maedche, A.: Ontology merging for federated ontologies on the semantic web. In: Proc. Int. Workshop for Foundations of Models for Information Integration (FMII-2001). (2001)
13. Ockerbloom, J.: Mediating among diverse data formats. Technical report, Carnegie Mellon Computer Science (1998)