ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Model for Digital Content Definition and Representation

Μοντέλο Ορισμού και Αναπαράστασης Ψηφιακού Περιεχομένου

Μουστάκας Σ. Βασίλειος

Επιβλέπων: Δελής Αλέξης , Καθηγητής Ε.Κ.Π.Α.

ΑΘΗΝΑ

Ιούλιος 2010

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Model for Digital Content Definition and Representation

Μοντέλο Ορισμού και Αναπαράστασης Ψηφιακού Περιεχομένου

Μουστάκας Σ. Βασίλειος

Α.Μ.: M913

ΕΠΙΒΛΕΠΩΝ:

Δελής Αλέξης , Καθηγητής Ε.Κ.Π.Α.

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

Δελής Αλέξης , Καθηγητής Ε.Κ.Π.Α.
Κολλιόπουλος Σταύρος , Επίκουρος Καθηγητής Ε.Κ.Π.Α.

Ιούλιος 2010

ABSTRACT

In the contemporary unified information and knowledge presentation field of the World Wide Web, there's an emerging need to represent complex structures, comprising individual semantic entities that can possibly act autonomously.

This work, introduces a generic and expressive model for information representation, which comprise information objects and relationships as well as properties they may have. This way we are capable of creating complex graph structures that can back the definition of modern, semantically rich, content hierarchies. Additionally, a dynamic type system has been developed for defining classes of information objects and relationships that determine, a common set of specifications to which, instances of that type must automatically conform, in terms of the properties they can have, relationships they are allowed to participate in and other uniform system-wide behaviour. The use of types, further refines our model'As expressive power and promotes reusability. Its capability to represent information is tested against reference use-case systems and the models they support.

We, furthermore, discuss the application of the aforementioned information model into a notional storage architecture, that will contribute to the homogenisation of diverse storage infrastructures and will enable the augmentation of current systems' expressive power, to match that of the model of this work. This functionality is exposed through a proposed general yet powerful programming interface with which the user can handle information access and retrieval as well as data definition. Finally, a brokerage mechanism is proposed, lying inside the architecture, which by exploiting the creation of a common storage resource pool, can be used to, transparently, distribute the information to the underlying infrastructure, based on user-defined factors, such as the cost, the available free resources or the data replication policies.

SUBJECT AREA: information management, storage management

KEYWORDS: information model, type system, information object, meta-data management, storage

ΠΕΡΙΛΗΨΗ

Στο σύγχρονο ενιαίο πεδίο προβολής πληροφοριών και γνώσης του Παγκόσμιου Ιστού, πυκνώνει η ανάγκη αναπαράστασης σύνθετων δομών, απαρτιζόμενων από επιμέρους οντότητες με σημασιολογία και ενδεχομένως αυτόνομη υπόσταση.

Στην εργασία αυτή, παρουσιάζεται ένα γενικό και εκφραστικό μοντέλο αναπαράστασης πληροφορίας, το οποίο αποτελείται από αντικείμενα πληροφορίας και σχέσεις καθώς επίσης και από τις ιδιοτήτες που ενδεχομένως έχουν. Με αυτό το τρόπο, καθίσταται δυνατή η δημιουργία πολύπλοκων δομών γράφων, οι οποίοι μπορούν να υποστηρίξουν τη μοντελοποίηση σύγχρονων, σημασιολογικά πλούσιων, ιεραρχιών περιεχομένου. Επιπρόσθετα, αναπτύσσεται ένα δυναμικό σύστημα τύπων για τον ορισμό κλάσεων αντικειμένων και σχέσεων, το χαρακτηριστικό των οποίων είναι ο καθορισμός προδιαγραφών με τις οποίες, αυτόματα, θα πρέπει να συμμορφώνονται οι οντότητες όσον αφορά τις ιδιότητές τους, τις σχέσεις στις οποίες μπορούν να παίρνουν μέρος και εν γένει τη συνολική συμπεριφορά τους στο σύστημα. Η χρήση τύπων ισχυροποιεί περαιτέρω το δοθέν μοντέλο αλλά και προωθεί την επαναχρησιμοποίηση των δομών του. Η ικανότητα του, τέλος, να αναπαριστά πληροφορία δοκιμάζονται σε αντιπαράθεση με περιπτώσεις υπαρχόντων συστημάτων αναφοράς και των μοντέλων που αυτά υποστηρίζουν.

Επιπλέον, συζητάμε την εφαρμογή του ανωτέρω μοντέλου πληροφορίας, σε μια ενδεικτική αρχιτεκτονική αποθήκευσης, η οποία θα συμβάλει στην ομογενοποίηση ποικιλόμορφων υποδομών αλλά και θα δίνει τη δυνατότητα αναβάθμισης των δυνατοτήτων αποθήκευσης των υπαρχόντων συστημάτων, ώστε να φτάσουν την εκφραστική δεινότητα και ισχύ του μοντέλου της εργασίας αυτής. Η λειτουργικότητα εκτίθεται μέσω μιας προτεινόμενης γενικής αλλά ισχυρής προγραμματιστικής διεπαφής με την οποία ο χρήστης θα διαχειρίζεται θέματα πρόσβασης και ανάκτησης της πληροφορίας, αλλά και ορισμού τύπων δεδομένων. Τέλος προτείνεται και ένας μεσιτικός μηχανισμός, εντός της αρχιτεκτονικής, ο οποίος εκμεταλλευόμενος την δημιουργία ενός ενιαίου χώρου συγκέντρωσης αποθηκευτικών πόρων μπορεί να χρησιμοποιηθεί για να διανέμει, διαφανώς, την πληροφορία στην υποκείμενη υποδομή σύμφωνα με παράγοντες ορισμένους από το χρήστη, όπως για παράδειγμα το κόστος, οι ελεύθεροι πόροι ή οι πολιτικές κατανεμημένης αναπαραγωγής των δεδομένων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: διαχείριση πληροφορίας

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: μοντέλο πληροφοριών, σύστημα τύπων, αντικείμενο πληροφορίας, διαχείριση μετα-δεδομένων, αποθήκευση

$\sigma\ \beta\ \varepsilon$

Σας ευχαριστώ

ACKNOWLEDGMENTS

First and foremost I would like to offer my sincerest gratitude to my supervisor, *Alex Delis*, Professor at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, for his invaluable help in writing this thesis and for giving me the freedom to work in my own way. Moreover, I attribute the level of my Masters degree to his continuous support throughout my post graduate studies and I would like to thank him for the trust so generously showed to me, so early, as well as for the knowledge, not necessarily always academic, that was amply willing to offer.

I am also humbly indebted to *George Kakaletris* for sharing his ideas with me, for all the fruitful discussions we had on aspects of this thesis, and beyond, and for giving me the opportunity to work with him in research projects in the area of digital preservation and retrieval.

Also I would like to mention the people in the Management of Data, Information and Knowledge Group (MaDgIK) at the University, for creating the perfect conditions for pursuing higher academic goals, collaborating in a competitive academic and working environment and most important trusting me with their friendship.

Last but not least, I thank all those close to me that had the patience and understanding to support and encourage me in all of my endeavours.

Contents

List of Figures

List of Tables

Listings

PROLOGUE

This work has been conducted and presented in partial fulfillment of the requirements of the diploma of Master of Science (M.Sc) in the Department of Informatics and Telecommunications, School of Science of the National and Kapodistrian University of Athens.

Chapter 1

Introduction

1.1 The Problem

The past few years the World Wide Web has witnessed an enormous explosion in digital content creation rate. This skyrocketing growth is certain to continue going, as all traditional types of media such as voice, tv, radio and print go from analog to digital; people continue to take pictures and video, send e-mail and tweet; companies are still adding to their data warehouses; Governments are still requiring more information be kept (24).

In 2009, besides the global recession, the digital content set a record, and grew by 60% in nearly 800 exabytes. Its growth continued, and 2010 the *zettabyte* (trillion of gigabytes) barrier has been surmounted. In 2011 the amount of information created is estimated to exceed 1.8 zettabytes, meaning that it is growing by a factor of 9 in the past five years (25). By 2020, the digital universe will be 50 times as big as it was in 2009, touching the inconceivable size of 35 zetabytes. This is translated into bits that will realize over 25 quintillion information containers; that is packages, files, images, records, signals e.t.c. On the other hand, the average size of these containers is getting smaller due to proliferation of embedded systems (sensor networks, smart houses e.t.c). This means that the manageable entities in the digital world are actually growing almost twice as fast as total number of gigabytes (24).

Most disturbing though, for informatics engineers, is that, along the next decade that the amount of digital information will be increasing 44-fold and the amount of digital containers 67-fold, the number

of IT professionals will be increasing by a slim factor of 1.4 (24)!

1.2 Importance of Research

Even though much progress has been made in the Web with search engines and semantic technology, the same cannot be said for storage architectures that fail to efficiently address these kind of challenges, often making it harder to find something on one's hard drive than on the Web. The above peculiar reality has a plausible explanation though. Hyperlinks among pages, convey invaluable information that can be proven very useful for searching and presenting results (42). On the other hand, classic filesystems can realize relationships among files, only through the hierarchical directory system.

Most of the information out there comprise unstructured data (e.g. images, voice packets). Adding structure to this unstructured data enables as to look beyond bit-streams and catch valuable information otherwise ignored by automated systems. For example, we can annotate the frames of a video with information about the people appearing in it. In fact the fastest growing type of data in the digital universe, described earlier, is *metadata*, or data-about-data (24). It is, thus, absolutely essential to deal with this information tsunami in new ways, others than simply building massive storage systems and dumping streams. We have to (re)think the ways we deposit and retrieve all this semantically enriched information.

1.3 Thesis Objectives

Our primary goal for this work was to design and implement a complete, yet generic, information model for defining, representing and efficiently manipulating diverse kinds of semantically enriched digital information.

Our secondary goal was the definition of a system architecture which would benefit from our information model in order to deliver sophisticated content storage services that would expose the following characteristics:

- effectively handle the ingestion of data information along side with extra description and

relationships between them

- leverage and homogenize diverse storage capacities and capabilities
- easily extended to support multiple storage models

The advantages of such an architecture are:

- augment the expressiveness of traditional persistency layers such such as file-systems,
- provide a unified storage resource pool independent of the specifics of the actual bit storage,
- cost effective resource aggregation since it will relay on currently available technologies,
- benefit from the maturity of the underlying storage technologies

1.4 Disposition

This report consist of five (5) Chapters and one (1) Appendix. Chapter 1 (this one) starts with a small description of the problem and motivation for this work. Moreover, there is an enumeration of the thesis objectives and a quick overview of its structure. Chapter 2 builds a momentum for the thesis by introducing basic concepts and reviewing the state of the art in the field. In Chapter 3 our proposed information model is thoroughly discussed and supported. Chapter 4 introduces our proposition for a reference storage architecture and elaborates on the features as well as added value it will convey. This work concludes with Chapter 5 by summarizing the important contributions and looking forward to future work. The Appendix A holds a discussion of a conceptual and practical prototype implementation of the our proposed information model.

Chapter 2

State of the Art

2.1 Review

In this chapter, we review some very important, research as well as enterprise work that has been conducted in the areas of content and storage management as well as digital preservation. Some has resulted in very successful products that have proven all (or at least most) of what they advertise they can do, in real world environments. Others, on the other hand, didn't make it so good. In any case, all have contributed in the area, and brought forward interesting new concepts and solutions to problems. The discussion focuses, mainly, on the information models they utilize in order to achieve their goals, since this is the basic topic for this thesis. However, whenever needed, additional information is given to help creating a more comprehensible picture for the given topic.

We made a simple categorization of the systems and specifications at hand, to further help the reader. We first look at the fundamental concepts and notions derived from the digital preservation model which offers key vocabulary for our later discussion. We then present how object exchange and reusability is promoted in the Semantic Web. Digital library systems play an important role in the review as they are considered pioneers in the field while they still preserve their position in innovation, handling the intellectual property of many, well known, institutes and universities around the world. Finally we touch upon research that has been conducted in filesystems to enhance the every day desktop experience.

2.2 Generic Models

This section discusses a couple of generic models for representing information as entities as well as collections of these entities. It may also be considered as a very good introductory point since we will first be reviewing the digital preservation model which was the product of very important research activity that introduced key concepts widely used today and will help in understanding the rest of the material.

2.2.1 Digital Preservation Model

In an ever changing Information Society, digital information is not destined to survive, unless there is some systematic process to preserve it, called *digital preservation*. This is contrary to real world artifact objects, such as books or celluloid strips, which exist until someone or something actively disposes them. One apparent reason of the ephemeral nature of data is their unreliable physical storage, like the volatile main memory of the computer or the difficult ``shelf life'' of magnetic tapes, hard-drives and optical disks. What is more important though, is the loss of data that is caused by the obsolescence dictated by the rapid technological changes, mainly in software applications.

There has been large international effort to address these issues. These efforts aim to provide standards for an exhaustive list of aspects for digital preservation in museums and libraries (57, 37, 29). Contemporary repository systems (30, 53, 50), adhere to the now dominant international standard of the OAIS Reference Model (19) and are an important component of modern digital preservation.

This part covers the fundumental concepts that revolve around the definition of an architecture for building specialized automated information systems, called *digital object architecture*. The core services of an implementation of such an architecture (the ``System'' from now on) are those that provide identification, access and management of digital assets.

Digital Objects

The basic form in which the *content* is stored and managed within the architecture, is called a *digital object*. The notion was introduced in (31) and has since been further developed and evolved (14, 15, 43, 17). Conceptually, digital objects are entities that provide (16): *i)* encapsulation,

ii) description, *iii*) unique identification, and *iv*) value-added access to content. The content is divided into *data* and data-about-data or *metadata*. This way multiple bits of relevant information, create a complex, multipart entity that acts and can be managed as a unity. If the data of a digital object is a digital object itself, then the former is called *composite*; otherwise *elemental*.

Types of Digital Objects

Digital objects are described by an abstract typing mechanism that provides a high-level, self-describing, type definition called *data type* (32) or *content type* (16). Content types are technical descriptions of a specific class of content that may also define intent-of-use based access to it. Thus, a digital object is an instance of some data type that encapsulates content, metadata or even access policies.

Saidis et.al (48, 49) propose an effective realization of *Digital Object Prototypes (DOPs)* which is a digital object type definition that provides a detailed specification of its constituent parts and behaviours. Its goal is to provide a mechanism that uniformly resolves digital object typing issues in an automated manner.

Disseminations

A digital object architecture must carefully distinguish and enforce the subtle but important differences between digital objects created by the owner of the data, digital objects stored in the System and digital objects accessed by the user (14).

A *dissemination* is the result of a request to access a digital object. We identify two classes of a disseminations: *i*) *primitive disseminations*, and *ii*) *content type disseminations* . The first are not extensible and, access the content of the digital object in a raw fashion, much the same way as it was deposited in the repository. Table 2.1 shows a basic set of primitive digital object disseminations. Since digital objects are usually complex entities though, we may not need to handle the whole object. It might be just some of its constituent parts defined for example either by the user's current needs (parameterising the request query) or by an access control policy. Such behaviour is achieved by content type disseminations, which provide views of a digital object the way its (content type) creator had imagined it to be accessed.

Function	Description
`CreateDataStream()`	Assigns a data stream to the digital object.
`GetDataStream()`	Returns the data stream from the digital object.
`DeleteDataStream()`	Removes a data stream from the digital object.
`ListDataStreams()`	Returns a list of data streams attached to the digital object.
`GetDissemination()`	Invokes a content type dissemination on the digital object.
`CreateDisseminator()`	Creates a content type disseminator.
`DeleteDisseminator()`	Deletes a content type disseminator from the digital object.
`ListDisseminators()`	Returns a list of content type disseminators for the digital object.

Table 2.1: Primitive digital object disseminations.

Repositories

A *repository* is a network-accessible storage system in which digital objects can be stored for future access or modification. There should be no limit on how many repositories can coexist in the architecture. According to (16) repositories should themselves be digital objects in order to facilitate additional functionality through disseminations.

The special user that owns the digital material is also part of the architecture and is called the *originator*. His role is to create digital objects out of that material and put them in the System in order to make them available to others. Such digital objects are called *stored* digital objects. If they can be modified after being placed in the repository they are called *mutable*, while if they cannot *immutable*.

A repository contains an *attributes record* for each of its stored digital objects. This record encapsulates all the extra information for the object that is either invariant for it over repositories or repository specific. Additionally, it may also have associated with it a transaction record, to record transactions of the repository involving the digital object.

Repository access protocol (RAP)

The repository must provide mechanisms for: *i*) creation, *ii*) deletion, *iii*) modification, and *iv*) dissemination of digital objects. This is accomplished through the implementation of a *Repository Access*

Function	Description
`VerifyHandle()`	Confirm whether the given handle is a registered handle
`AccessRepoMeta()`	Access the repository metadata
`Verify_DO()`	Returns true the a digital object with the given handle is stored in the repository
`AccessMeta()`	Access the metadata for a specified digital object
`Access_DO()`	Return a dissemination of the given digital object
`Deposit_DO()`	Store a digital object in a repository
`Delete_DO()`	Purge a digital object from a repository
`MutateMeta()`	Edit the metadata for a digital object
`Mutate_DO()`	Edit a digital object

Table 2.2: Basic RAP operations

Protocol (RAP). A simple set of service requests are shown in table 2.2 (16).

Handle infrastructure

A digital object is globally identified by a unique string, called a *handle*. Handles are produced by a *handle generator* and are expected to be published to the System though well known *handle server*s. Digital objects with published handles are identified as *registered*.

Architecture Layering

Arms et.al in (15) proposed a three component structure of a repository that comprise 2.1: *i*) a *persistent store* which is the storage space (database, filesystem e.t.c.) used to hold the information, *ii*) a *repository shell* which is the interface to the outside world (by implementing some RAP) and is usually designed to work with a very wide range of persistent stores, and *iii*) an *object management* component that implements the logic for mapping digital objects to a specific store.

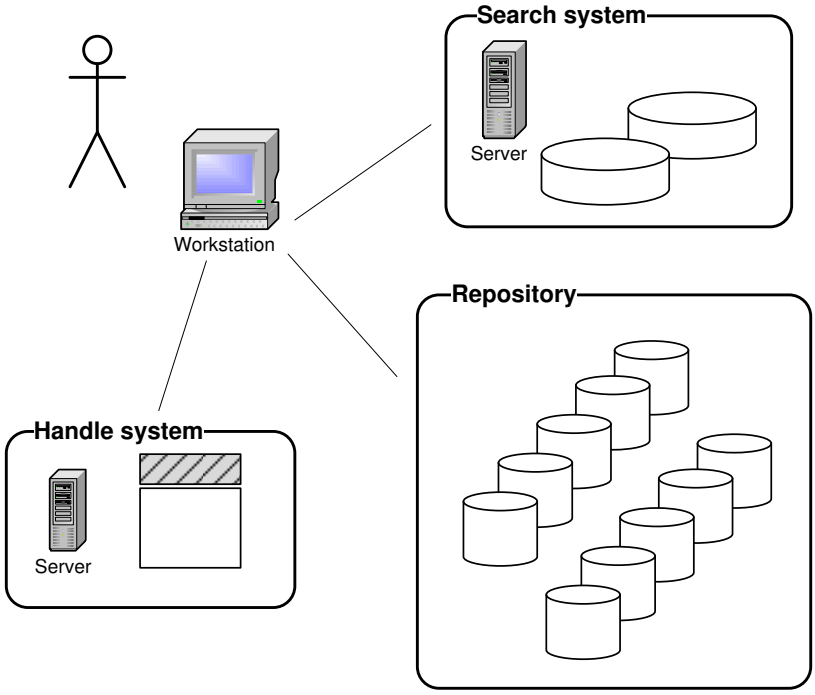


Figure 2.1: Major system components.

2.2.2 Open Archives Initiative - Object Reuse and Exchange (OAI-ORE)

Open Archives Initiative has detected a lack of a standardized way to identify the constituents as well as the boundary of an aggregation of Web resources. Thus, a project was born, called Object Reuse and Exchange (10, 35, 33) that gathered international experts from publishers to e-Science communities to identify, profile and develop an extensible specification that would enable repositories, agents, and services to interoperate in the context of use and reuse of compound digital objects beyond the boundaries of their holding repositories. This interoperability (58) will yield more efficient and effective: *i*) object discovery, *ii*) object (and ``part-of'') reference *iii*) view dissemination *iv*) object aggregation (disaggregation), and *v*) agent-oriented processing .

OAI-ORE builds on top of the following foundations:

- WWW Architecture (28)
- Linked Data Effort and the Semantic Web (18)
- Resource Description Framework (RDF) (40, 39)

OAI-ORE has been already positively welcomed from the research community, and early efforts has been put on the adoption of the specification on top of already stable and mature digital preservation architectures (56). That way digital objects are decoupled from repository software used to manage them. This way a set of low-level resources are introduced, which are then managed by a series of independent services including repository software such as EPrints (6, 54, 52), Fedora (53).

OAI-ORE Data Model Entities

The primary feature of the OAI-ORE abstract data model is expressing relationships between Web resources. In this part, we introduce the basic model entities.

Aggregation An *Aggregation* is a Resource that is a set of other Resources. The specification uses *URI-A* to denote a URI of an Aggregation. Since an Aggregation is itself a Resource, it can be an Aggregated Resource in another Aggregation resulting in nested Aggregations. Relationships may be asserted between the Aggregation and/or constituent Aggregated Resources and information external to the Aggregation. These external resources may be individual Resources or Aggregations themselves (i.e. citations, translations).

Aggregated Resource An *Aggregated Resource* is a Resource that is a constituent part of one or more Aggregations. In a respective manner, *URI-AR*s denote URIs that identify Aggregated Resources. Relationships may be asserted between Aggregated Resources denoting for example containment semantics of a Resource over another or that one is an alternative representation of the other. Aggregated Resources may also have have types (e.g. bibliography or table of contents) defined by various vocabularies.

Resource Map (ReM) A *Resource Map* is a Resource that: *i*) describes a single Aggregation, *ii*) asserts the finite set of constituent Aggregated Resources, and *iii*) may express types and relationships between the Aggregation and its Aggregated Resources. The specification uses *URI-R* to denote URIs that point to Resource Maps and must be different from the URI-A of the Aggregation it describes.

Proxy A *Proxy* is a Resource used optionally to ``stand for'' an Aggregated Resource in a specific, to the respective Aggregation, way. Similarly, A Proxy is also described using a URI referred to as *URI-P*. Sequencing is an example of what can be achieved via Proxies. In applications such as scholarly communication there is a need for a stronger relationship, indicating lineage. This is also achieved via Proxies.

OAI-ORE Data Model Relationships

OAI-ORE standardises the description of the relationship between digital objects. A ReM asserts a set of RDF triples expressing information about an Aggregation, its Aggregated Resources, metadata about either the Aggregation, the Resource Map or both, as well as other Relationships. For example a ReM must include a triple with an `ore:describes` predicate. The subject of this predicate must be a URI-R for the ReM, while its object the URI-A of the Aggregation it describes. Other types of relationships concern metadata about ReMs, As, Aggregated Resources, relationships between an Aggregation and similar resources (or even other resources and types), nested Aggregations and Proxies.

Representations

RDF/XML (39) is the most popular way for serializing OAI-ORE ReMs, though there are others, like Atom/XML (26).

The essence of OAI-ORE is depicted in figure 2.2 and can be summarized as follows. An Aggregation is introduced so that we can unambiguously refer to a collection of other Resources. It has a URI (just like any Resource on the Web) and it does not have a Representation since it is a conceptual construct, according to Semantic Web Resources. An additional Resource is introduced, that of a Resource Map, which also has a URI and, in addition, a machine-readable Representation that *i)* expresses which Aggregation it describes (`ore:describes` relationship in Figure 2.2), *ii)* lists the resources that are part of the Aggregation (`ore:aggregates` relationship in Figure 2.2), and *iii)* express relationships and properties pertaining to all these Resources, as well as metadata pertaining to the Resource Map itself (`dcterms:creator` and `dcterms:modified` relationships in Figure 2.2)
.

2.3 Digital Repositories

Repositories are an important aspect of content management. They are alternatively referred to as digital preservation systems or digital libraries[1]. Their basic characteristic, though, is that they conform

[1]Mainly because the first prototypes had libraries as their application domain.

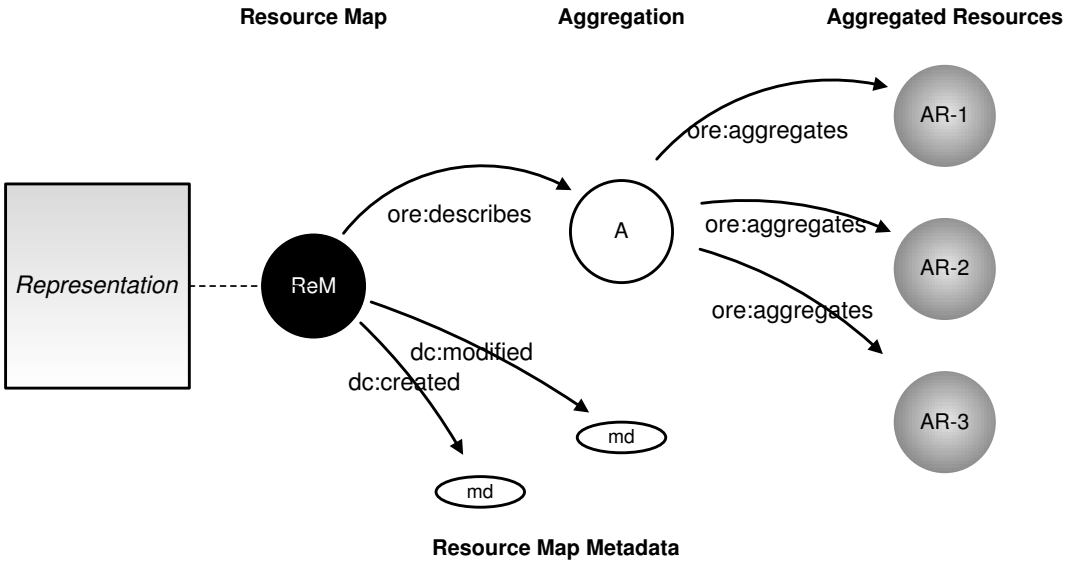


Figure 2.2: The Aggregation *A* aggregates three Resources *AR-1, AR-2, AR-3* and is described by Resource Map *ReM*.

to the Consultative Committee for Space Data Systems' Reference Model for an Open Archival Information System (OAIS) (19).

Common challenges of these digital repositories are scalability, interoperability with other repositories, and efficient work-flow support for ingestion of large numbers of digital objects. Some important issues to be addressed in structuring information can be found in (15). For example, digital objects are frequently related to each other through relationships. It is also very likely that the same object might be stored in several digital formats or, in other words, provide alternative representations. In any case though, all data is given an explicit data type. Moreover, all metadata is encoded explicitly and do not rely on any semantic information provided to interpret them. Versioning might be also important in such systems as digital objects are, inherently, highly volatile entities. An other important issue is that each digital information entity might need to be handled differently according to security access permission attributes associated with it.

2.3.1 Flexible and Extensible Digital Object and Repository Architecture (FEDORA)

Flexible and Extensible Digital Object Repository Architecture (FEDORA) (53) is a modular architecture built on the principles of interoperability and extensibility and implemented as a set of web services (with WSDL described APIs). For this reason it can co-exist within a larger web-service framework

providing access and management for digital preservation in a variety of multi-tiered environments. This, significantly differentiates FEDORA from other existing vertical, all-in-one systems (50, 1, 6, 8).

Digital Object Model

Fedora uses a *compound digital object* abstraction to aggregate one or more data items into one manageable entity (34, 59). These data items can be stored in a local repository or in a remote one. It comprises (see figure 2.3):

- a unique persistent identifier (PID),
- system properties for managing and tracking the object, and
- one or more datastreams

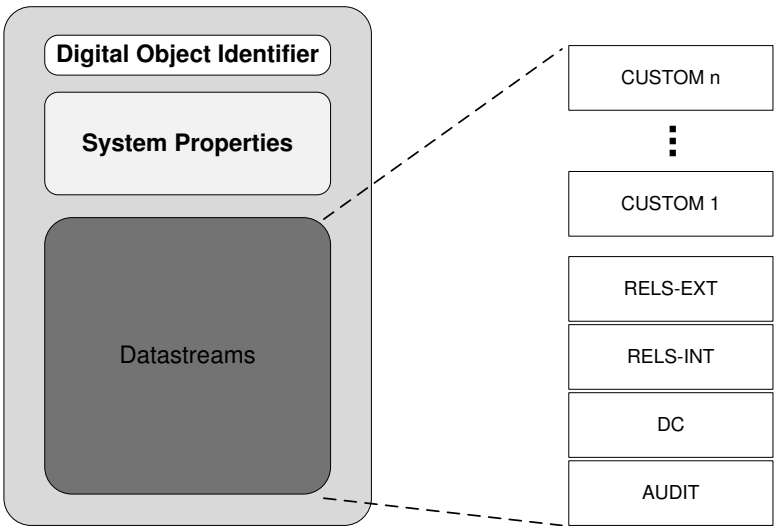


Figure 2.3: A depiction of the Fedora digital object model.

*Datastream*s aggregate data items. Each datastream is given a unique identifier within object's scope (along with a series of other datastream specific properties). FEDORA reserves four ids, namely:

- *AUDIT*: for maintaining auditing trails for the given object,
- *DC*: for storing metadata for it,

- *RELS-INT*: for describing internal relationships among the object's datastreams, and
- *RELS-EXT*: for external relationships with other digital objects

Content Model Architecture

Even though every Fedora digital object follows the aforementioned model, there are four distinct kinds of objects that can be ingested in a Fedora repository:

Data object is the simplest, most common of all types of objects in Fedora. It is used to represent a digital content entity as we have already defined it above.

Content model (CModel) object is a special control object which acts as the container for the Content Model. This formal model characterizes a type of digital objects as well as specify the kinds of relationships which are either mandatory, permitted or forbidden between digital object groups. Thus all object types in Fedora, are organized into CModel object classes. An object belonging to some class asserts the relation `hasModel` with the corresponding CModel object. Inheritance from multiple CModel objects is supported and such a Data object should conform to all of its Content Models, containing an aggregation of all the datastreams defined in each of them. Fedora automatically assumes that all objects conform to a system-defined Basic Content Model.

Service definition (SDef) object is a special control object that contains the model of a service. An SDef defines just the interface for someone to interact with the object (but not exactly how this is performed). SDefs are associated with Data objects, indirectly, through a `hasService` relationship asserted by the Data object's CModel object. A Data object (through its CModel object) may support more than one Service by having multiple SDef relationships. The Fedora repository defines the Basic Service Definition which are the Operations shared by all objects (e.g. a function that provides direct access to the datastreams).

Service deployment (SDep) object is a special type of control object that describes how a specific repository will deliver the Service Operations described in a Service Definition object for a class of Data objects described in a Content Model object.

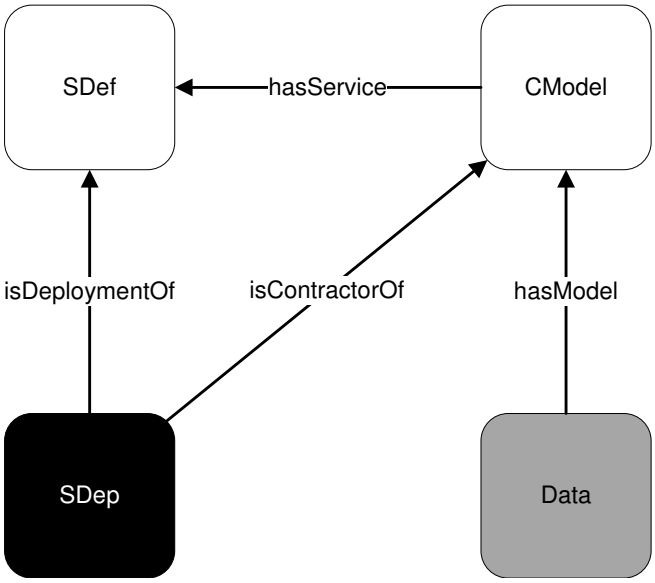


Figure 2.4: Fundamental content model architecture relationships.

Digital Object Relationships

Fedora digital objects can be related to each other in a variety of different ways. Digital object relationship metadata is a way of asserting these various relationships. These metadata are encoded in XML using the Resource Description Framework (40, 39) and stored in specially reserved datastreams in an object. A default set of common, generic relationships for creating complex digital object graphs is defined in the (7). These relationships can be refined or extended as well as create whole new arbitrary ontologies to encode relationships among Fedora digital objects tailored to specific needs.

2.3.2 DSpace

DSpace (50) is an open-source software platform for capturing, storing, indexing, preserving, and redistributing the digital assets of an organization. Its code-base is a joined effort by MIT and Hewlett Packard for use on Windows and Unix/Linux platforms and is distributed under the BSD open source license, which enables users to customize or extend the software as needed. It is written in Java using an array of free open-source software such as PostgreSQL, JDBC, Lucene e.t.c. It uses the CNRI Handle System (2) for assigning, managing, and resolving persistent identifiers for digital objects, a qualified Dublin Core vocabulary derived from the Library Application Profile (5) for

Source	Target	Source	Target
Data	CModel	hasModel	Identifies the class and, optionally, the object containing a model of the essential characteristics of the class.
CModel	SDef	hasService	Identifies the object containing a model of the functional characteristics of class members.
SDep	SDef	isDeploymentOf	Identifies the object containing a model of the functions being deployed.
SDep	CModel	isContractorOf	Identifies the object containing a model of the information being deployed.

Table 2.3: The supported relationships between fundamental object types in the CMA.

common description across all content types, the METS (9) standard for information packaging, and a Harmony/ABC model-based mechanism for recording the history of changes within the system.

Data Model

DSpace's information model (4) has been designed to easily adapt to the structure of the organization on which it is deployed (see Figure 2.5 for depiction and Table 2.4 for an example). Each such site is, thus, divided into *communities* and further divided into *subcommunities*. Communities contain *collections* that can be thought of as groupings of related content. A collection may span into several communities. Collections comprise of *items* that are considered the basic archival elements of the archive. Each item is owned by one collection but may appear in others too. Items are further subdivided into *bundles* of bitstreams. A *bitstream* is some data representation (usually ordinary computer files). Closely related bitstreams are organized into bundles that may, optionally, have a primary bitstream. Finally, each bitstream is associated with one *bitstream format* which is a consistent way of referring to a particular file format.

Note that the model enables multiple inclusion at all levels, from communities to bitstreams.

Metadata

DSpace holds three kinds of metadata about archived content (55):

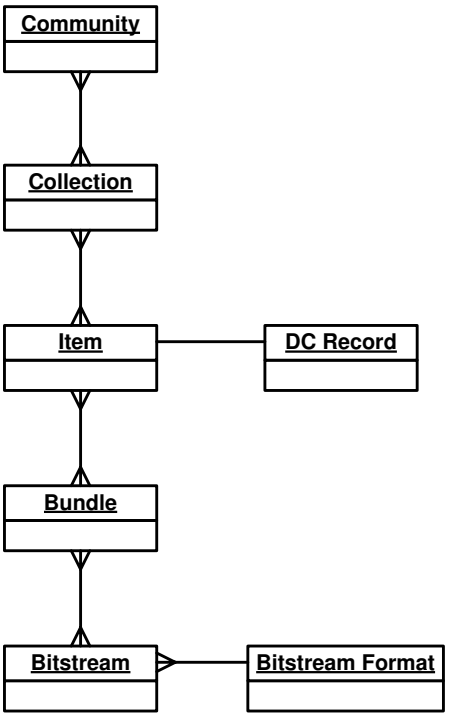


Figure 2.5: The DSpace data model.

Descriptive DSpace is able to support any flat metadata schemas (e.g. Dublin Core's Libraries Working Group Application Profile, or other user defined schemas).

Administrative Includes preservation metadata, provenance and authorization policy data.

Structural Defines how to present either an item itself or bitstreams within an item, to an end-user, and the relationships between constituent parts of the item.

Layer	Example
Community	Faculty of History and Archaeology (subcommunity of UoA's School of Philosophy)
Collection	History Collection
Item	Reports on Word war II
Bundle	HTML Files, Digitized videos
Bitstreams	a video clip
Bitstream Format	H.264/MPEG4 Advanced Video Coding

Table 2.4: An illustrative example of the DSpace data model.

Only three fields are required: title, language, and submission date, all other fields are optional. There are additional fields for document abstracts, keywords, technical metadata and rights metadata, among others. This metadata is displayed in the item record in DSpace, and is indexed for browsing and searching the system (within a collection, across collections, or across communities).

2.3.3 D4Science

D4Science (the follow-up phase of Diligent) is a European e-Infrastructure project which continues the vision of GÉANT and EGEE projects towards the realization of networked, grid-based, and data-centric e-Infrastructures that accelerate multidisciplinary research by overcoming barriers related to heterogeneity, sustainability and scalability (3). This requires, among many others, the support of a disciplinary repository that can effectively handle and preserve digital content. D4Science's Content and Storage Management services have to bear this weight and the following is a quick discussion of the information model that supports them.

D4Science's information model comprises two elementary constructs (see Figure 2.6 for an Entity-Relationship model depiction):

- information objects, and
- object references

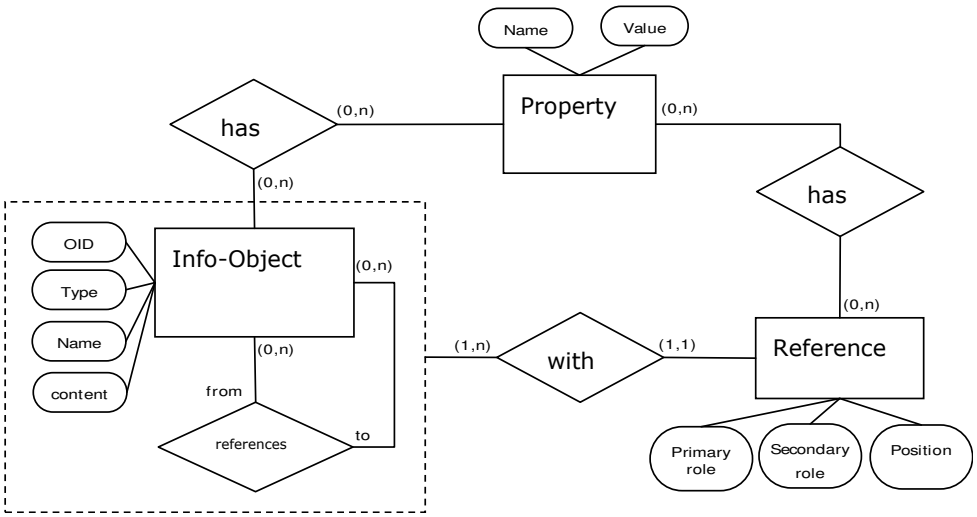


Figure 2.6: ER model for D4S' information model.

Information object model

The information object represents the elementary information unit. It is described by some key properties, namely: *i*) object identification (OID), *ii*) name, and *iii*) type (document collection, aggregate, metadata, external document e.t.c.) . It can be annotated by a number of additional custom properties and can be associated with raw content. Complex metadata can be represented as information objects that are associated with the object they describe via appropriate relationships.

Relationship Model

An object reference is a link that associates two information objects. It supports directed $m - n$ relationships. There is the ability to label object references with a two-level type attribute in terms of `primary role:secondary role`. The secondary role is optional and is used for further type specialisation. For example `is-member-of` type reference indicates that the source object is a member of the collection target, or `is-described-by` indicates that the target takes the role of metadata for the source object and therefore describes the source. As an example of a secondary role `is-described-by:is-annotated-by` indicates that the target takes a more specific role than before, that of an annotation, which has slightly different semantics from metadata in D4Science (see figure 2.7).

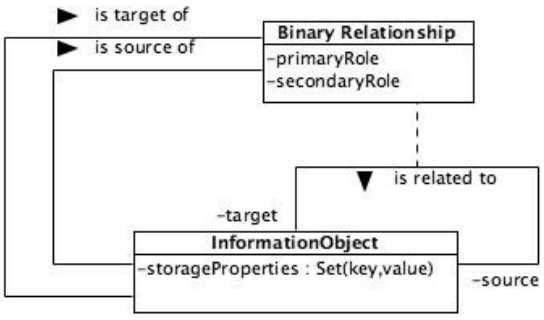


Figure 2.7: The basic relationship model in D4S.

The dependency notion among relationships and sources (targets) means the existence of the later depends on the existence of at least one target (source). Also a relationship is called exclusive if it associates its source (target) to exactly one target (source), otherwise it is called repeatable.

Document Model

A document is a compound information object. It consists of information objects linked together via appropriate relationships (e.g. HTML page with its images). A document can exist only as a part of a collection.

Collection Model

Collections are the basic data structure used to organize content inside the information organization services of D4S, by aggregating documents. Members of a collection share enough similarities to be homogeneously processed. It is described by a Collection identifier, for easy reference, and a number of specific properties. Moreover, collections can be nested, which means complex networks of interconnected information can be easily structured, static if objects are added and deleted explicitly or virtual if current members are determined dynamically at access time through declarative membership predicates.

Metadata Model

It is a type of relationship with primary role `is-described-by`. It gives to targets the semantics of metadata about the corresponding sources. The source of such type of relationship is a metadata object for the corresponding target. The model supports both metadata objects and metadata collections. `is-described-by` relationships are exclusive for their targets (a metadata object corresponds to one object) and repeatable for their sources (an object can have an arbitrary number of metadata objects).

Annotation Model

A secondary role `is-annotated-by` is applied to the primary role of `is-described-by`, which gives its targets the semantics of annotations for the corresponding sources. Expected targets for such a relationship are documents, but it is not a requirement.

2.4 File Systems

Traditional file-systems, such as ext3/4 and ntfs, store a file as a plain *stream of bytes* and have limited information about the data inside those files. Such systems provide a predefined set of simple object metadata, mostly maintained by the operating system and held in directories and file control blocks (e.g. inodes). In these systems apart from assigning file names, users can effectively specify metadata by: *i*) creating a directory hierarchy, *ii*) defining file extensions, *iii*) encoding metadata in the filename, *iv*) putting metadata as comments in the file, or *v*) maintaining adjunct files related to primary data files . This approach has obvious defects and limitation. For example, take the path `grad.dit/pms515/2010/s1/ass1/913/sync.c` which assigns the following attributes to the file `sync.c`:

- *school* > graduate at Department of Informatics and Telecommunication
- *course* > Advanced Operating Systems
- *year* > 2010
- *semester* > 1
- *studentId* > M913
- *assignmentNum* > 1
- *filename* > sync
- *filetype* > C source

The searching capability of the above is limited, for the attributes are stored hierarchically, and accessed via a path specification. For instance we can easily locate all student assignment submission for a specified course, while we are unable to find all course assignments that a particular student has submitted within a given semester.

2.4.1 Extended Attributes

A better solution is to tag files with information that describes them. This information is known as *property metadata* and it adds description to plain data. This allows files to be searched is new

ways, not previously possible such as ``finding pictures having person X''. Property metadata can be handled by the file system either by itself, in a native manner, or through some extension.

Experimental Metadata-Enriched File Systems

Some experimental file system implementations fall into the first category, which are self described as inherently metadata-enriched storage systems. They maintain user-defined attributes as an intrinsic part of the file data and are capable of associating relationships among files while in the same time provide an advanced metadata querying scheme (11, 12, 13).

Linux Extended Attributes

More down to the ground approaches fall into the second category of filesystems handling extended attributes through an extension. All widely known filesystems from the world of free/open source and proprietary software have an implementation of such functionality. In Linux for example, the ext2, ext3, ext4, JFS, ReiserFS, XFS, Btrfs and OCFS2 1.6 filesystems support extended attributes (abbreviated `xattr`) if the kernel is compiled with the `libattr` feature enabled.

Extended attributes are not widely used in user-space Linux applications, even though they are available since kernel version 2.6[2].

2.4.2 Forks

In a computer filesystem, a *fork* is byte stream associated with a file system object. Thus, every non-empty file must have to be associated with at least one fork. Depending on the filesystem, a file may have one or more other associated forks, which in turn may contain primary data integral to the file, or just metadata. The difference between forks and extended attributes, is that forks can be of arbitrary size, possibly even larger than the file's primary data fork.

[2]Beagle and Dropbox are known to be using extended attributes, though.

Alternate Data Streams (ADSs)

NTFS introduces the notion of *Alternate Data Stream (ADS)*. This technology allows more than one data streams to be associated with a filename. The namespace format is `filename:streamname` (i.e. `Bob_Dylan_-_Like_a_Rolling_Stone.mp3:lyrics`). With Windows 2000, Microsoft started the use of ADS in NTFS to store things like author names, document tiles and image thumbnails. Windows NT versions include the ability to use forks in the API[3], but mainly they are ignored by most programs, including Windows Explorer and the `DIR` command. With Service Pack 2 for Windows XP, Microsoft introduced the Attachment Execution Service that uses alternate streams to enhance security. Extra metadata stored in ADSs may be displayed in the Windows Explorer as extra information columns, with the help of an Active-X component that is able to identify them. Windows Explorer copies forks and warns when the target file system does not support them, but only counts the main fork's size and does not list a file or folder's streams. The `DIR` command has been updated in Vista to include a switch for listing the extra forks.

Resource Forks

File system forks are associated with Apple's Hierarchical File System (HFS). Apple's HFS, and the original Apple Macintosh File System (MFS), allowed a file system object to have several kinds of forks.

Data fork The actual main data.

Resource fork Designed to store non-compiled data that would be used by a GUI (i.e. localisable text strings. However the feature was so flexible that additional uses were found, such as dividing a word processing document into content and presentation and then storing each part in separate forks[4].

Multiple named forks One of HFS Plus' ambitious concepts which has gone largely unused until Mac OS 10.4 which added a partial support implementation for Apple's extended inline attributes.

[3] Some command line tools can be used to create and access forks.

[4] As compiled software code was also stored in a resource, often applications would consist of just a resource fork and no data fork.

Risks in Forks

When a file system supports different forks, the applications should be aware of them, or security risks can arise:

- The user is not aware by the presence of alternate streams since they are not listed in the file browser nor are included in the file's size calculation.
- Malware software is known to be able to use alternate data streams to hide its code (38).
- Data can be lost when sending files using fork-unaware channels (e.g. via the e-mail service, or copying between file systems with no fork support[5].

2.4.3 MIME Types

Modern file systems are assisted by MIME types. This is achieved by embedding in the MIME type, metadata for the content that is actually stored on the disk. For example the popular *Nautilus* file manager of the *GNOME* desktop environment, uses MIME types to:

- open the file in an appropriate application
- display a string that describes the type of file
- display an appropriate icon to represent the file
- display a list of other applications that can open the file

2.4.4 Desktop Search and Related Technologies

Desktop search applications (i.e. Beagle, Spotlight) take the above concept even further by extracting data, including attributes, from files (using a file-format-specific filters) and indexing it. This enables searching based on both the file's attributes and the data in it.

[5]In the case of the two file-systems being fork-aware, the same must occur for the software that performs the actual copy, as well!

Google Desktop is a desktop search software for Linux, Mac and Windows platforms. After installation of the software, Google Desktop completes an indexing of all the files in the computer. After the initial indexing is completed, the software continues to index files whenever this is necessary. Google Desktop can inherently handle several types of data from e-mails to OpenDocument, Microsoft Office documents and several multimedia formats. Additional file types can be supported through the use of plug-ins. Google Desktop has received extensive critisism on several issues concerning security and user privacy infringement.

Microsoft Search also referred to as *Instant search* is Microsoft's approach on desktop search tools. Upon installation, it builds a full-text index of the files on a user's hard drive. Searches are performed not only on file names, but also on the content of the file. The later is provided by implementing the `IFilter` interface for the corresponding file types. Once an appropriate `IFilter` has been installed for a particular file format, it is then used to extract the text from that kind of files. Windows Search by default includes handlers for common filetypes such as Excel spreadsheets, PowerPoint presentations, HTML documents, text files, MP3 and WMA music files, JPEG images, among others.

2.4.5 Advanced File Systems

However, this still lacks the ability to handle related data, as disparate items do not have any associations defined. For example, it is impossible to search for ``the skype names of all persons who live in Athens and each have more than 100 appearances in my photo gallery and with whom I have had e-mail within the last week''.

In order to leverage the above problems engineers have come up with the idea of storing data in filesystems as files and handling annotations, relationships and other metadata inside relational database management systems. Filesystems' ability to efficiently store large amounts of data combined with databases' quering superiority, resulted in an interesting hybrid solution.

Windows Future Storage (WinFS)

Windows Future Storage (WinFS) (41, 60) was the name of a canceled Microsoft project for a data storage and content management system that was planed to integrate into *Longhorn* platform of technologies. It was based on a file-ystem and relational databases hybrid schema and designed for

persistence and management of structured, semi-structured (e.g. images, videos) and unstructured data. Unfortunately WinFS had weaknesses itself, such as increased design complexity, reduced performance, and difficult to retain consistency between its two components, leading to the project's discontinuation. Although WinFS was put into cryogenic freeze by Microsoft in 2006 (47), some of its components found their way into other major Microsost projects such as ADO.NET's *Entities* (21) and SQL Server's *FileStream* and *FileTable* (23).

Architecture Overview

From an engineering viewpoint, WinFS is made up of the following layers (bottom-up approach of figure 2.8) (46):

Core Implements the essential functionality someone would expect from a file system (e.g. Win32 access service, acl e.t.c.).

Data Model Leverages the functionality exposed by the core to provide added value item structuring, relationships and others.

Schemas Backs the reuse of information across applications.

Services Further extent the system functionality with capabilities like advanced querying or network synchronisation.

API Exposes, programatically, the advanced features of WinFS and even leaves it open for extension.

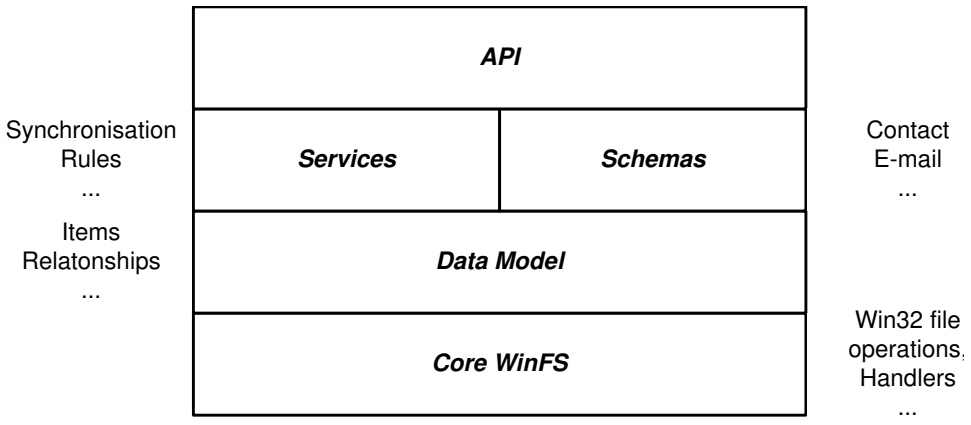


Figure 2.8: The WinFS architecture layering.

WinFS allows the manipulation of any type of information, provided that a well defined schema of that type has already been defined. Thus any application ``knows'' how to handle the data, making it reusable system wide. Moreover relationships can be asserted between individual data items. This can be achieved either in an automatic inference mechanism by the system (e.g. based on the file system hierarchy, or a file's access control list) or explicitly by the user. This enriched information structure enables advanced searching and accessing capabilities.

Types

WinFS recognises the types of data it stores. It does not handle them just as arbitrary bit streams like other filesystems do, but as instances of these data types. WinFS provides four base types, namely:

- *Items*,
- *Relationships*,
- *ScalarTypes*, and
- *NestedTypes*

The fundamental building block of types is the *property*. An Item can have properties that can be either a ScalarType meaning the simplest possible information units available (e.g. `Integer` or a `Date`), or a NestedType which defines a complex information unit consisting of two or more ScalarTypes (i.e. `E-mail`) (see figure 2.9). The type system closely relates to the .NET framework's notions of class and inheritance. You are, thus, able to create new types to be handled by WinFS by extending an already available one. WinFS promotes data reusability by making data types accessible to any application as well as their schemas so that the later can ``read'' the structure and be able to access the portions of the data they are interested in without the need of extra, ad-hoc utility programs written by the programmer.

WinFS creates tables for all defined types (61). The fields of the type are columns in the table and instances of the type are rows[6]. The database back-end allows SQL-type queries, XPATH searches (20) as well as the use of Microsoft's *OPATH*, a query language designed for handling directed acyclic graphs (45).

[6]For example, for semi-structured information, structured data are all filling the table in the relational store while unstructured are kept to a traditional persistence store.

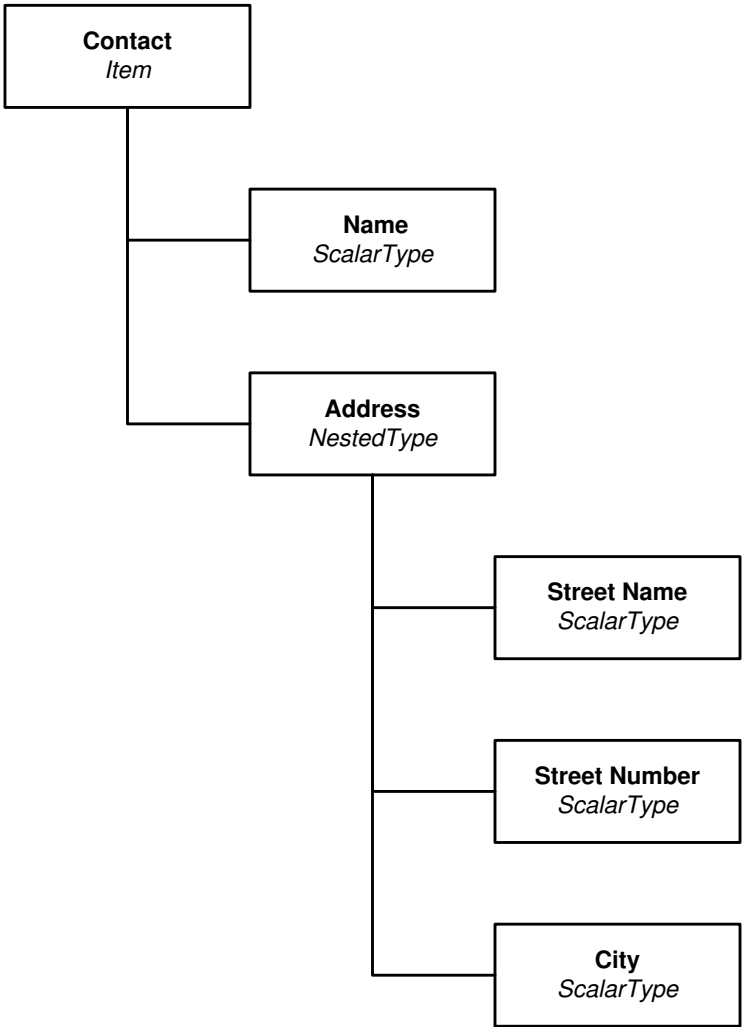


Figure 2.9: An example of an `Item` in WinFS.

Relationships

Moreover, WinFS can relate data, say an `E-mail` and a `Contact` by a `From` relationship. This is done once more with the use of properties. The `E-mail` will contain a `From` property and during access the relationship will be traversed and the related data returned. WinFS supports one-to-one relationships as well as one-to-many between items and data. Since many items can also relate to other items, a web of information is actually formed, creating many-to-may relationships.

Relationships are realized by extra fields that refer to a row in a different table. The schema of the relationship specifies which tables are involved and what is the kind and the name of the relationshp (27). WinFS specifies three different primitive relationship types from which the programmer can

extend:

Holding relationship specifies ownership and lifetime of the target item. When the source is removed, the target is also removed. Because the target item can be a target for more than one holding relationship, it is removed when all the source items have been removed.

Reference relationship provides simple linkage between two items, but each item will continue to be stored even without the other.

Embedding relationship expresses hierarchical relationship between a parent item and a child item.

Rules

WinFS includes a mechanism for handling automatically events that have to do with items and relationships (22). For example if a condition concerning the attributes of two items is met, a relationship might have to be automatically created. Rules are exposed programmatically as .NET objects like items and relationships and thus can be extended and augmented.

Chapter 3

A Generic Information Model

3.1 Information Model

An *information model* is the conceptual foundation of any data manipulating system. It identifies the basic entities and defines a set of rules among them in order to support, in a systematic way, the representation of the information within a system, enforcing the functionality is meant to expose. It must be relatively simple, in order to be easily applicable, but in the same time sufficiently expressive to fully back the rationale of the given operational needs as well as open to extensions for future ones. The benefits of conceptual modeling comes from creating and preserving stability. For example, often contents change but the application remains the same. In other occasions, is the other way around. So, it is obvious that the content needs to be separated from any kind of restrictive context that obstruct its preservation and thus shortens its life span.

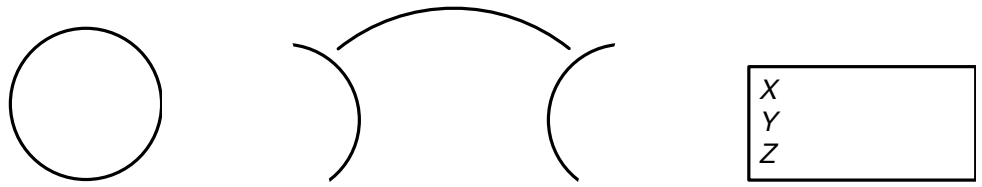
3.2 Instantiable Entities

Our model comprises the following primitive tools:

- Information Objects,
- Relationships, and

- Property Values

Figure 3.1 shows the graphical representations of these tools, a convention that will be followed for the rest of this chapter. Information objects are depicted as empty cycles (Figure 3.1(a)), relationships among information objects as links (lines) between them (Figure 3.1(b)) and property values as rectangles (Figure 3.1(c)) that enclose a set of such values and are sticked on the corresponding entity.



(a) An information object. (b) A relationship. (c) A set of propertie values.

Figure 3.1: The information model's inhabitants.

With these model tools at hand, the system architect has to imagine everything in terms of information objects, relationships and property values. This creates a rich information network like the one showing in Figure 3.2. Graph-based models are known tools in the literature for having very strong semantic expressiveness in impressing complex conceptual concepts ``in a form that is logically precise, humanly readable, and computationally tractable'' (51).

3.2.1 Information Objects

An *information object (IO)* is the entity that represents any valuable data asset in the system's environment.

An information object can hold:

- property values that provide meta-data information for the artifact it represents,
- relationships that outline the associations of the information object with other ones, and
- the digital content that constitutes the object, either by encapsulating the raw data or simply preserving its reference to the storage system.

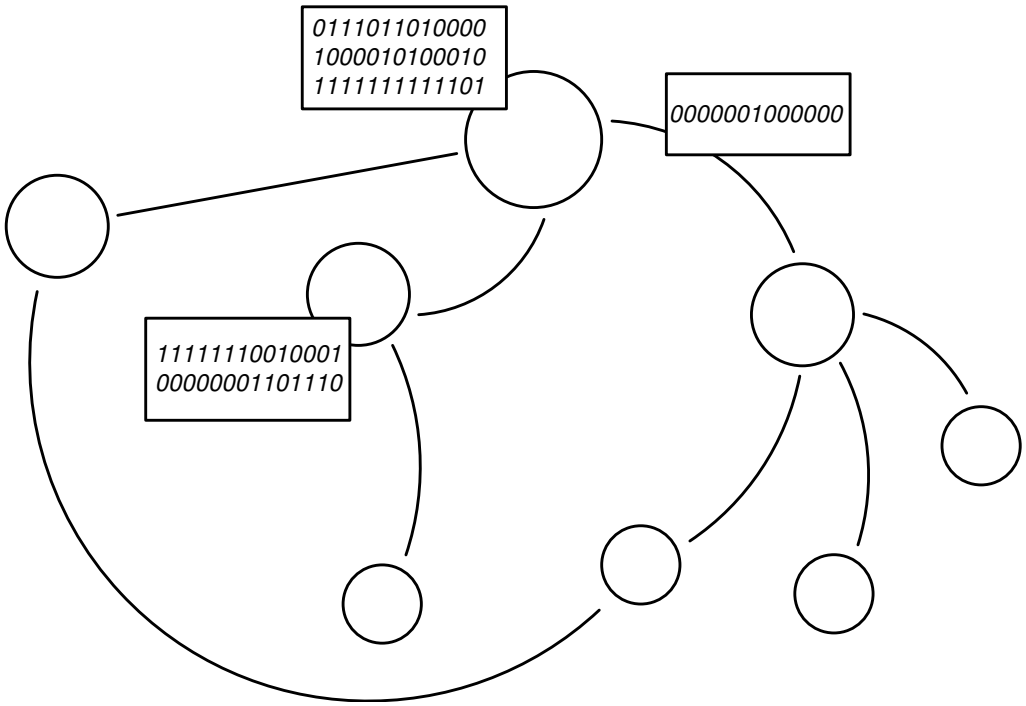


Figure 3.2: A graph that corresponds to some data.

Examples of information objects include, but are not limited to, an `Article`, a `Document`, a `Book` with `Page`s, an `Image`, a `Song` etc.

3.2.2 Relationships

A *relationship (R)* is the entity that implements an association between two objects.

A relationship can contain property values that further enhance its semantics.

With relationships, we are able to represent complex concepts, specified by means of *part-of* (e.g. *hierarchical*) relationships between information objects, or*associative* (e.g. *non-hierarchical*) relationships between information objects. Examples of such instances could be, `employs` connecting an `Employer` with an `Employee`, or `compiles` connecting a `Music Album` with a `Song` in its track-list.

3.2.3 Property Values

A *property value (PV)* represents a value of some descriptive, technical and/or administrative meaning.

An information object may contain property values; a relationship may contain, as well. A property value on the other hand, cannot be described by other property values.

Property values can be *dynamic*, if the value of the property they represent is computed just-in-time to reflect a constantly updated facet of the system, or *static* if it must be assigned and/or updated explicitly.

3.3 Type System

Remember figure 3.2? This is a pretty looking draw, with a lot of curly geometrical shapes all over, creating some kind of graph. This would have been a description given from anyone that is clueless of the real context in which this representation lives. It is obvious that it actually conveys little information to anyone other than its creator. Even for the later, it will be very difficult to understand it after some time.

In computer science, a *type system* can be described as "a tractable syntactic framework for classifying phrases according to the kinds of values they compute" (44). According to (36), the use of type systems yields the following essential features:

1. Provide an efficient programming error detection mechanism before runtime.
2. Serve as a data structuring tool for design and modeling.
3. Provide a sound maintenance framework.
4. Give sufficient information for optimization purposes.

Even though the above are discussed in a programming language context, the ideas are applicable anywhere we need such features. All programming languages have to define their safe and sound object model themselves, after all.

3.3.1 Thinking in Types

When we say that an object is of some *type*, we implicitly refer to a class of objects that are described by a certain set of properties and expose certain kinds of behaviour when interacting with other objects of the same or other type. Having already introduced the model's basic citizens, now there is a need to think a little bit about the notion of a type.

Creating a type system, allows us to take advantage of the semantics of these types in order to implement logical constraints such as, which properties and of what value type an instance of a specific class can or must carry, or which types of information objects, a specific relationship type, is logically sound to link together.

A type can be thought of a set of properties. Each property comprise: *i)* a property name, and *ii)* a property value type . The later defines the kind of values this property can hold. In order to define types we need to answer a question like ``what are the essential characteristics, say, of an image?''. Thus, type is used to classify objects, not contain real world data or values, like shown in Figure 3.3.

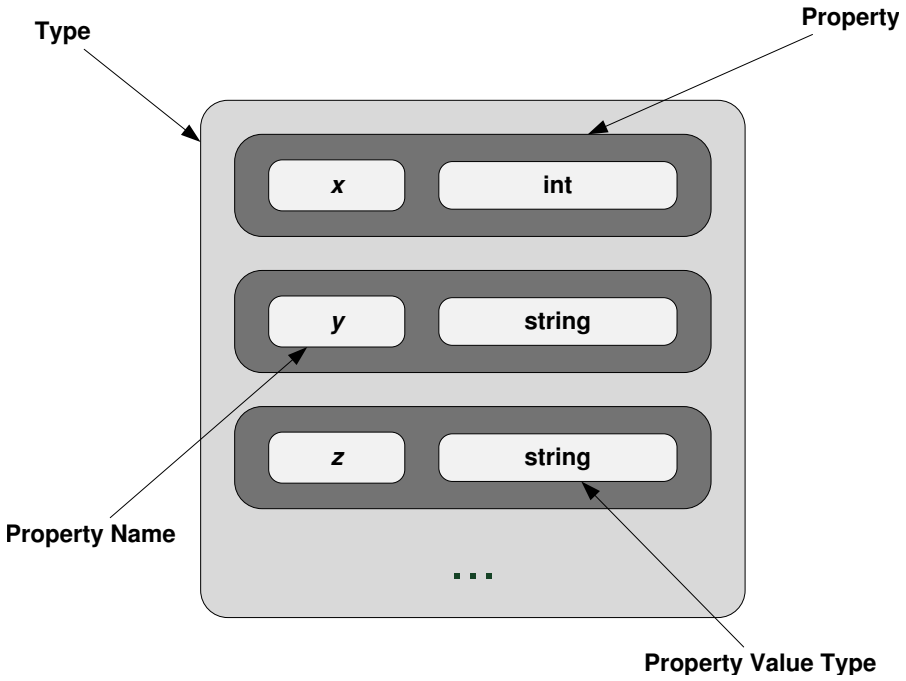


Figure 3.3: A sketch of a type comprising some property value types.

On the other hand, an instances are created by supplying values to the properties defined in their corresponding type.

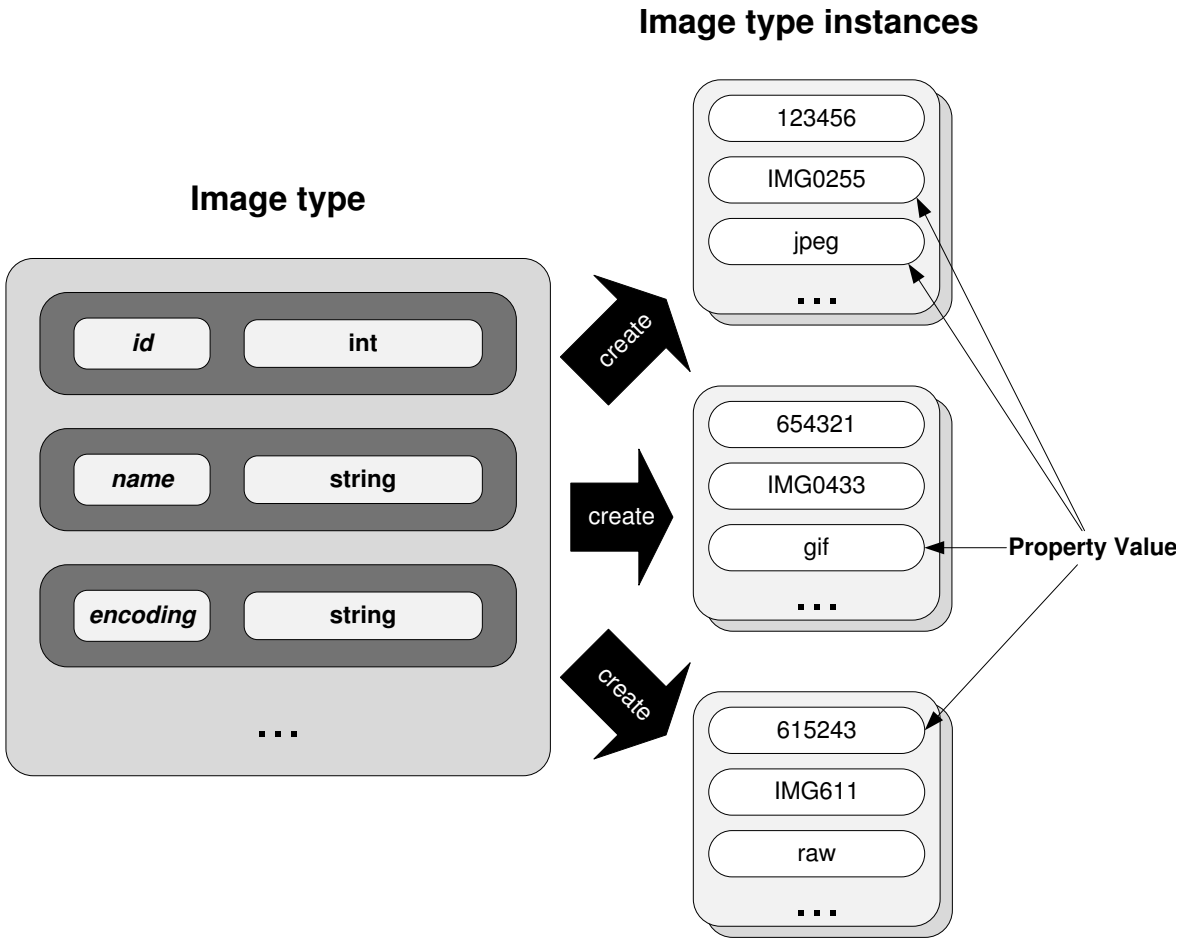
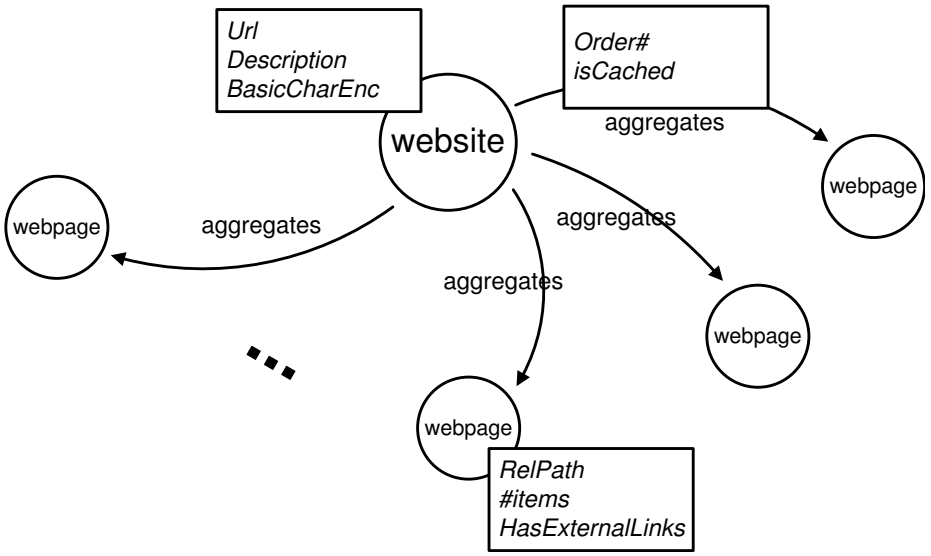


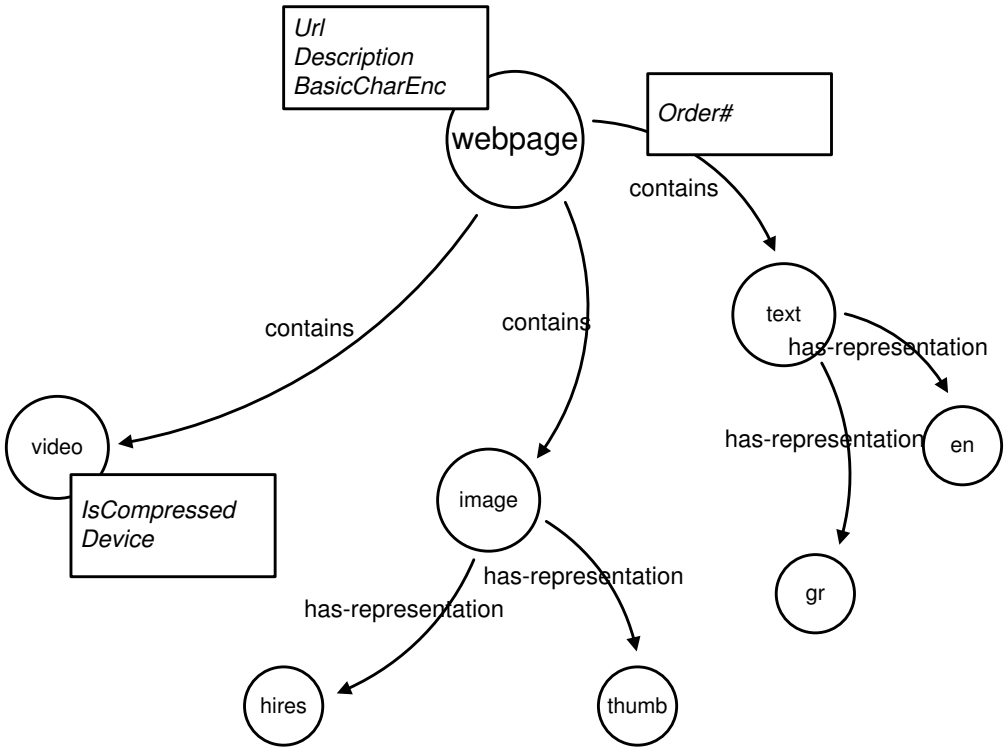
Figure 3.4: Instantiation of an *Image* type in three information objects.

In an object-oriented design approach another part should be added here; that of the behaviour of some type of entities. In this work, we leave behaviour outside the discussion, since we confront the design of this information model, primarily from the perspective of a system, providing generic storage that has some added value over traditional persistency technologies, rather than a full-fledged content management system.

Let us now review the problem of figure 3.2, using a simplified real-world example. Suppose we want to map the information in a website to our model. The easiest conceptual interpretation of a website is a multistage ``assembly'' of elementary information entities. At a first stage, a `Website` can be thought of as a collection of texttttWebpages. Each `Webpage` is by itself a container of any combination of `Text`, `Image`, `Video` or `Audio`. Figure 3.5 depicts the above interpretation in terms of information objects that are related to each other through associations denoting that a `Website aggregates Webpage`s.



(a) The `Website` `aggregates` `Webpage`s.



(b) A `Webpage` close up.

Figure 3.5: A simplified website representation using our typed information model.

Note how much more information is conveyed by this new graph. We can, now, easily distinguish those information objects that are webpages and those that are videos or text. All webpage information objects have property values that are not unknown sequences of bits any more. They denote a relative path under the website or a boolean value that says if it contains links that point outside of the webserver. Relationships are also differentiated. Aggregations for example do not have an order number which means the order in which we aggregate information objects is insignificant. Also being arrows, instead of simple lines, relationship depict the flow of association, specifying which of the linked information objects is the source and which the target.

3.3.2 Information Object Types

An *information object type (IOT)* defines and describes classes of information objects with respect to their attributes and interaction with other elements.

It can be be a subclass of one or more parent types, thus implementing a multiple inheritance hierarchical information object type system.

We apply rules, namely:

- property rules, and
- relationship rules.

3.3.3 Relationship Types

A *relationship type (RT)* defines and describes classes of relationship objects with respect to their attributes and the interaction they have with linking information objects.

A relationship type can be a subclass of one or more parent types, thus implementing a multiple inheritance hierarchical relationship object type system.

Relationship types are called *directed* if they involve a semantically clear flow of information from an explicitly defined *source* information object to an explicitly defined *target* information object. The

part, that an information object type plays in a relationship (whether it is the source or the target), is referred to as its *role*.

A relationship type can also be either *loose* or *tight*. In the first case, the relationship instance can continue to exist if one of the connected objects is deleted. The second type is further subdivided into *cascaded-deletion* and *deny-deletion*, where deletes are cascaded all the way down the relationship chain or denied to be performed, respectively. For example an `is-aggregated-by` relationship is tight with deny-deletion on its target in order to enforce the behaviour that an object is deleted only if it is deleted from all the aggregations that it takes part. Further such behaviour, for tight relationships, can be added in a system's business logic if it is needed. For instance another tight relationship could dictate that if it is established then none of the participants is allowed to change (e.g. the relationship between an invoice and the corresponding bank transaction for its payment).

We define the following sets of rules:

- property rules,
- role rules, and
- multiplicity rules

3.3.4 Rules

Types, as we have already mentioned, are tools used to define a family of objects that share the same properties and interactive behaviour with other objects inside the context of our system. For example, to have a valid `Image` instance, there should be a mandatory property, named `mime` storing the MIME type of the payload it conveys, or a `Thumbnail Image` should also have some restriction in resolution. Moreover, we could need to ban objects from participating in specific relationships, because for example a `File` cannot be the source of a `contains` relationship or we could narrow the accepted types of participants in a relationship, because a `Collection` object should `aggregates` $0..n$ other objects, or a `JPEG Collection` should `aggregates` $0..n$ other *JPEG* objects.

A *rule* is the mechanism we propose here in order to specify such policies and later enforce them into the system. We define three kinds of rules, namely: *i)* property, *ii)* relationship, and *iii)* role rules.

Property rules can be applied to either entity type (information object type or relationship type) and define the set of properties the entity type can or must have and their corresponding property value types.

pr 1. The entity can have the specified property name.
(i.e. an `Image` object can have a property `application` to be able to store the application name with which the image has been created).

pr 2. The entity must have the specified property name.

pr 3. The entity must not have the specified property name.

pr 4. The entity has the specified property name which can have the specified property value type.

pr 5. The entity has the specified property name which must have the specified property value type.

pr 6. The entity has the specified property name which must not have the specified property value type.

pr 7. The entity has the specified property name which can have the specified property value type with the specified property value.

pr 8. The entity has the specified property name which must have the specified property value type with the specified property value.
(A `JPEG Image` object has a property `mime-type` of `string` type and value `image/jpeg`).

pr 9. The entity has the specified property name which must not have the specified property value type with the specified property value.

pr 10. The entity can have a property name with the specified property value type.

pr 11. The entity must have a property name with the specified property value type.

pr 12. The entity must not have a property name with the specified property value type.

Relationship rules can be applied to any information object type to define in which relationship types can it participate in. Its role is left to be determined using the role rules found within the corresponding relationship types.

rer 1. The information object type can particite in the specified relationship type.

rer 2. The information object type must particite in the specified relationship type.

rer 3. The information object type must not particite in the specified relationship type.

Role rules can be applied to any relationship type and define which information object types are valid sources or targets for it[1]:

ror 1. The relationship type can link the specified information object type either as source or target.

ror 2. The relationship type can link the specified information object type as a source.

ror 3. The relationship type can link the specified information object type as a target.

ror 4. The relationship type can link the specified information object type X as a source if the target is of information object type Y.

ror 5. The relationship type can link the specified information object type X as a target if the source is of information object type Y.

ror 6. The relationship type must link the specified information object type either as source or target.

ror 7. The relationship type must link the specified information object type as a source.

ror 8. The relationship type must link the specified information object type as a target.

ror 9. The relationship type must link the specified information object type X as a source if the target is of information object type Y.

ror 10. The relationship type must link the specified information object type X as a target if the source is of information object type Y.

ror 11. The relationship type must not link the specified information object type either as source or target.

ror 12. The relationship type must not link the specified information object type as a source.

ror 13. The relationship type must not link the specified information object type as a target.

ror 14. The relationship type must not link the specified information object type X as a source if the target is of information object type Y.

ror 15. The relationship type must not link the specified information object type X as a target if the source is of information object type Y.

Multiplicity rules can be applied to any relationship type:

[1] If we need to have a directed relationship type and thus explicitly define the source and the target.

mr 1. (source, target) The relationship is exclusive on its source (target) and can be related to exactly one target (source).

mr 2. (*, target) The relationship is repeatable on its target[2].

mr 3. (source, *) The relationship is repeatable on its source.

mr 4. (*, *) The relationship is repeatable system-wide.

There is an obvious redundancy between relationship rules and role rules. It lies on the fact that we define the same thing in two different places. In a relationship rule, for example, we can specify in which relationship types the container information object type is allowed to participate while in a role rule we specify which information object types act as source or target in the container relationship type. This redundancy can be prone to conflicting relationship and role rules but allows ease of definition and leaves the system architect free to decide for the resolving mechanism that will be used[3].

3.3.5 Property Value Types

Property values are not exactly an integral part of the object model we propose. It can be thought of as ``quick'' meta-data for the system. No hierarchy is assumed in this model for Property Value Types. They can be one of the following primitive types:

- boolean
- double
- float
- integer
- long

or of the non-primitive type

[2] An `is-aggregated-by` relationship is repeatable to its target if we assume that an object can be a member of more than one aggregation.

[3] A resolving decision could be that role rules (since they are defined in the relationship type itself) are more important so conflicting relationship rules are ignored.

- string

Structures, introduce difficulties such as the unclear ordering semantics they provide. For example, ordering a collection of arbitrarily sized images by their $widthxheight$ pixel size properties, is actually not a straightforward process.

Property assignment is a safe approach since storage properties are more static and not so volatile. Such assignment will probably be hard-coded inside implementing classes.

3.4 Evaluating the Information Model

Here we quickly discuss, the ability of the information model we just proposed to map to other models we review in Chapter 2. Thus, one representative from each category of approaches was taken in order to show how its basic notions and structures can be replicated in order to be represented by our model.

3.4.1 OAI-ORE

We chose OAI-ORE as the representative of generic models of Section 2.2 for its brevity, clear target and its emerging influence among the the various Semantic Web technologies.

The mapping of OAI-ORE to our model is quite straight-forward. The notion of an aggregation is similar to that of a collection so we can represent it as a complex information object that does not encapsulate any content by itself. It collects other information objects that contain the content via typed `aggregates` relationships.

These later objects are themselves a type of Resource in the OAI-ORE model. We can associate them with content, either by encapsulation of raw content within the object or applying a specially typed relationship.

Resource maps as well as Proxies are also special types of Resources in the ORE model, so we can map them easily as specially typed information objects.

Last, the relationship typing system we have introduced, is capable of creating any semantic representation we need for relationships defined by the OAI-ORE system such as `ore:describes` or `ore:aggregates` or any that is used by it such as `dc:modified` or `dc:created`.

3.4.2 D4Science

The D4Science's gCube information model presented in Section 2.3 among other repository implementations is used here because it is very similar to our model and is collection-oriented as OAI-ORE following the dictates of this Semantic Web for easy access of content and metadata.

The information object, document and collection models of GCube can be represented with our tools as simple and compound information objects. Document and Collection can aggregate their constituent data via appropriate relationships whose semantics and behaviour can be easily tailored to the specific needs by implementing and extending types of relationships.

The Metadata model can be mapped either by creating information objects with assigned metadata properties or applying `is-described-by` typed relationships between a primary information object and a metadata information object. Using relationship type subtyping, relationships of type `is-annotated-by` can be created be inheriting and then extending the semantics of `is-described-by` typed relationships in order to enable the representation of the GCube's Annotation model through the use of our model.

3.4.3 A File System

In Section 2.4 we review the capabilities of contemporary file-systems that are exposed directly from them or through some extended architecture on top. Here we first attempt to prove the capability of our information model to map the basic file-system entities and structure and then how we could extend them to handle more complex data.

Strict aproach

The fundamental file system entities can be mapped easily with our model. Containers and File-system Items are identified as the most basic among them, thus a set of predefined information objects can be offered, such as:

- Disk
- Directory
- File

creating the hierarchy depicted in figure 3.6. These objects *i)* statically implement the semantics of the basic file-system notions, and *ii)* convey straight-forward business logic (e.g. storage handling embedded in the code)

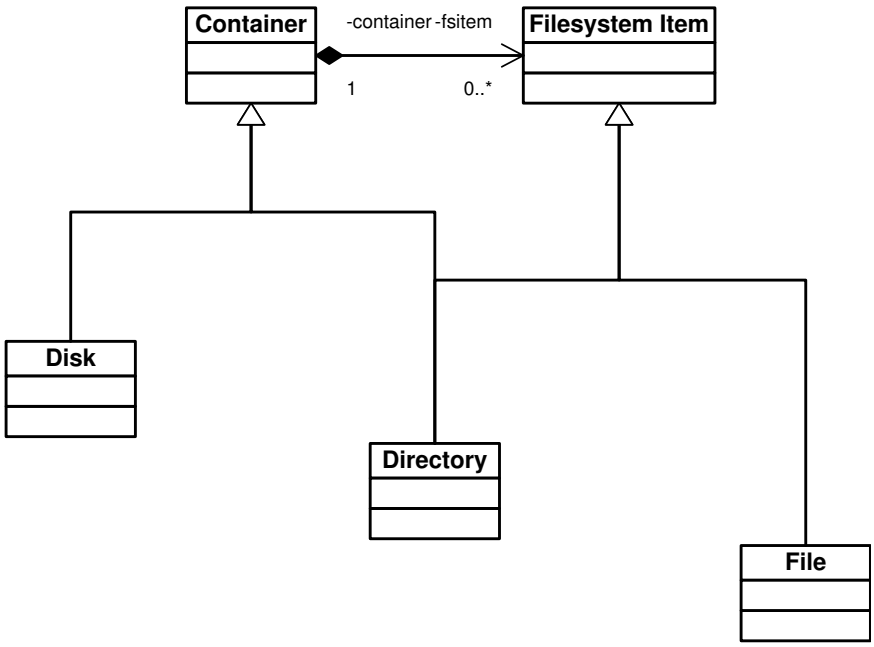


Figure 3.6: Mapping a file system on our information model.

A couple of examples for relationship types could be `contains` for container typed objects like Disks and Directories in order to gather file system entities (see Table 3.1) and `links` for implementing simbolic links.

Source	Target
container	file
container	container

Table 3.1: A relationship description for `contains`.

Loose/variable approach

We could extend the above schema in order to enrich the expressiveness of traditional file-systems. According to a variable implementation, the file system could be used as the storage facility for raw contant as well as for some of the basic properties it can handle itself. This could depend on the capabilities of the underlying technology. Extra data of structured information could be stored either on a small backing RDBMS or even serialized on the file-system in a fast and inteligent way in order to be easily retrieved so that the system could deserialize the complex information object. Another key issue is to store these metadata in a comprehensive way for other applications in order to enable easy exchange of information independently of our API.

Chapter 4

Meta S+0R3 A Reference Storage Architecture

4.1 Vision

The greater vision behind this thesis work, is to create a full system storage architecture that will be based on the information model we propose and describe in Chapter 3, in order to achieve a twofold purpose.

- Enable the homogenisation of the storage technologies available in underlying persistency layers.

- Elevate the capabilities of underlying storage architectures to reach the expressive richness our information model is capable to deliver.

We propose and describe here, in this Chapter, a reference system architecture for storage we call *Meta S+0R3* (read: meta store), yielding the above essential features, that target modern applications which require full and efficient manipulation of complex information entities.

4.2 Proof of Concept

Using traditional storage architectures, such as common disk arrays and other, more advanced, data management systems, like relational database management systems, can yield some very important positive effects. Their *simple installation*, *straight forward configuration* and relatively *cheap cost per gigabyte* for storage hardware, all lead to very good capacity scaling. Subvened by modern approaches in data modeling (i.e column stores) and processing (i.e. map-reduce) this vast number of byte streams can efficiently be handled and manipulated according to each need. All these, with the additional ensurance provided by the *maturity* of those systems, which are exhaustively tested in diverse application domains and environments. Unfortunately, this maturity comes from the fact that these technologies belong to an older generation, that has not been equipped with tools, to cope with the needs created by the contemporary Information Society we currently live in. Thus, we find poor handling of metadata enriched representation, ineffective data organisation and inefficient search mechanisms.

On the other hand, recent approaches, particularly from the world of digital preservation, allowed the world to store and retrieve enriched representations of previously plain data that conveyed much more extensive semantic information than before and thus more accurately imitating the complex intellectual endeavours of mankind. This way information retrieval has been made more sophisticated and thus more efficient and accurate. However, many times, specific operational environments and application domains have been targeted thus making them difficult to configure for other application cases, if not unable at all. Moreover, license restrictions are also common to such huge software applications meaning that they are closed platforms themselves or tied to some proprietary software or hardware dependency (e.g the storage technology).

The pros and cons of the above paths look complementary. Why not try to fuse them together, then? This way we could try to overcome each others' negative aspects by using and enforcing their positive elements and practices.

The rationale is to create an architectural layer that would

- homogenise several models,
- aggregate plain storages, data management and content management systems to provide a unified resource pool,

- be released from the shackles of specific underlying storage infrastructure,
- augment the expressiveness of traditional (robust and mature) persistency layers
- leverage and homogenize diverse storage capacities and capabilities

4.3 Architectural Description

The block diagram of the architecture is depicted in Figure 4.1. The Meta S+0R3 architecture consists of three basic modules: *i*) a *global reference point*, *ii*) a *management layer*, and *iii*) an *adapting layer* . Additionally, there is the the persistency layer, which is part of the architecture but outside its main functinal modules

Following, is a top-down elaboration on each of the consistuents of the architecture.

4.3.1 Storage Pool

A very important observation on Figure 4.1 is that the actual storage layer, the *storage pool*, is outside Meta S+0R3, depicting the intention to use the storage infrastructure as is, without performing any kind of ad-hoc implementation at this level, or requiring anything to change to some proprietary or specific technology. This way storage systems retain their autonomy by securing that data reach this part with constructs that are understandable by the given persistence technology (e.g. a HDFS or a Grid FTP). Additionally, this openness means that other applications, outside our system's border, that have a clear understanding of the semantics of the information persisted, are able to benefit from it by using them on their own.

Several storage options can be available

- traditional file-systems
- relational database management systems
- other types of storages (repositories etc)

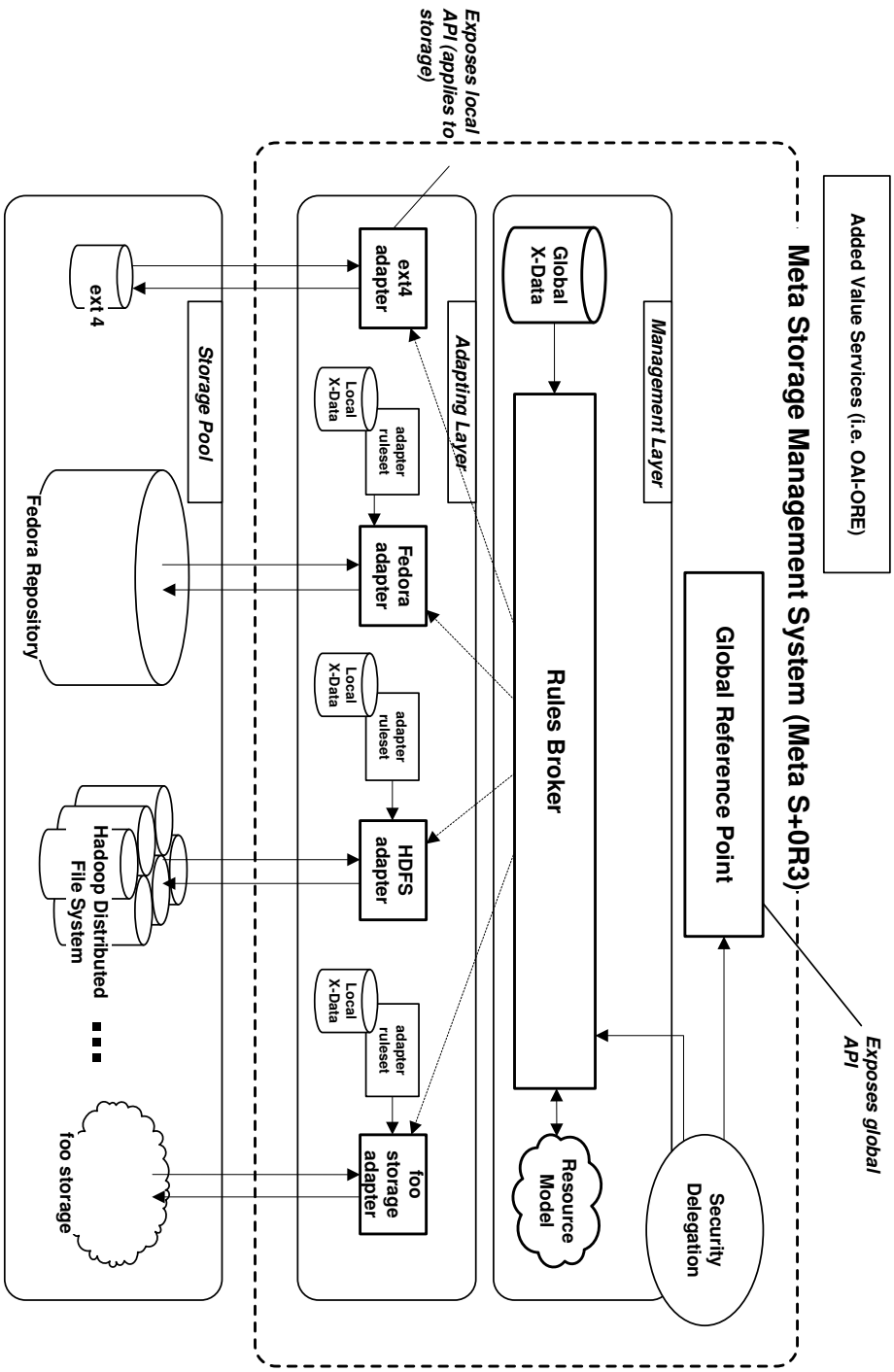


Figure 4.1: The proposed architectural envision of a meta content management storage system.

4.3.2 Adapting Layer

We could just use the expressiveness of any storage technology available, but this does not give any added value to a system. The need is to enable the storage of enriched information structures. The *adapting layer* is a software layer on top of the storage that helps to achieve the basic goals of the architecture, namely *i*) homogenizing storage technologies, and *ii*) extend their expressiveness.

Some storage systems are more expressive than others. Using the proposed typesystem, we can leverage and homogenise diverse storage capacities and capabilities.

Any lacking functionality could be added explicitly by the corresponding adapter for the specific underlying storage. To achieve this, the adapter can have

- its own storage space beside it (for example a RDBMS) in order to save any additional data the system has to handle but cannot be directly stored with constructs offered by the underlying layer, and

- a set of norms, that dictate how things are treated below the adapter.

The first is shown in Figure 4.1 as *local x-data* while the second as *adapter ruleset*. An example of a local x-data entity could be an RDBMS. An adapter ruleset can contain rules dictating, for example, that extra metadata are encoded in a XML file using a specific nomenclature and this file is put into a hidden folder besides the raw content named `.metadata`.

Depending on the functionality that is given to us, inherently, by the underlying technology, as well as the expressiveness we want to expose, this extra software layer can be either extensive, small or be absent altogether.

The adapting layer should expose, at least, a basic *local API* that is able to handle the information in the corresponding storage technology. In the background, the adapter will use the mechanisms described earlier in order to ingest and then be able to expose enriched data without the caller needing to know how the data were restated and then restructured to be returned.

4.3.3 Management Layer

The model that is exposed by the adapting layer can be lifted even more with a *management layer*. Since the system independently stores information regardless of the actual underlying persistency technology, this layer could provide us with additional added value services. If we can ``speak'' the model of one of them, the homogenization feature actually enables us to to do it the same way for all, transparently to the application requesting a content service. This creates a space that connects the individual storage elements, all together. We can then map one to another. Can this be done directly? If it cannot be done perfectly, is there a possible way to downgrade the content somehow, to fit the target storage, in order to process the content and then bring it back again, without losing any of the expressiveness needed by the application? These are questions answered by the management layer.

The main component of this layer is the *broker*. The broker decides when and how to distribute the data to the underlying adapting layer for specific storages. This could be a rules based broker making decisions based on factors like:

- Capacities/capabilities with respect to our proposed information model
- Storage
- Co-location/proximity
- Performance constraints
- Application restrictions
- Replication needs
- Cost

The rules could be as simple as a routing norm. For example, an entity is due to go to a predefined storage because of a specific metadata it carries (i.e. all documents authored by Vassilis should be stored in a special storage entity). Or it could be more sophisticated ones, like replication and operation in heterogeneous environments. This could be done based on the availability, the optimal distribution of storage load or specific application restrictions. So it could, ultimately, transfer the content to grid nodes, when they are ready to process it or transfer it along a multi-node Hadoop cluster when mapping and reducing.

The broker needs to have a clear view of the potential of the underlying capabilities and exploit them, in order to make the right decisions. To do this, it needs to have some kind of *resource model* in order to know what lies beneath it, and some implementation specific *global extra data* needed to take advantage of the resource model in the best, most efficient way possible. For instance, norms defining the steps and/or any transformations needed to be performed in order to go from one storage to another. Thus, when a data entity arrives for storage, and is simple enough to be digested by an ext4 file-system, then this is the destination the broker can choose. Otherwise, if the information conveyed is more complex, then another storage is searched, either being inherently capable of storing such complex data or is enabled by our adapter implementation.

Moreover, additional mechanisms could be added at this level, such as security or access policies for the distribution of content across the storage pool.

4.3.4 Global Reference Point

The services of the Meta S+0R3 architecture are made available through a *global reference point*. This component exposes an API similar to that exposed in the adapting layer augmented with policies, which might be instructions for the management layer on how to treat a specific call. Also it might have additional methods for the system's introspection which are not significant to the model, or the system's definition. For example, registering extra repositories or defining brokerage rules etc.

The global reference point may also backed by other mechanisms such as security.

4.4 Basic API

In this section we propose a basic application programming interface for the Meta S+0R3 architecture. It is, for the time being, a non-object-oriented API. Both API's from the global reference point and the adapter layer are similar. The extra functionality foung in the former is that it gets a set of additional policies of how the call is to be treated by the management layer.

4.4.1 Aceess and Retreive

This part of the API is to perform basic access and retrieve calls.

`createInformationObject()` creates an information object.

`storeInformationObject()` persists the Information Object to the storage.

`locateInformationObjects()` returns an array of object GUIDs by taking as input a set of requirements (e.g. the type, properties carrying, name.

`retrieveInformationObject()` takes a GUID and returns an Information Ooject.

`retrieveCompleteInformationObject()` takes a GUID and a set of requirements (i.e. relationship navigation termination rules) and returnes a complete graph of an Information Object

`removeInformationObject()` takes a GUID and removes the corresponding Information Object from the storage.

`getInformationObjectProperties()` takes the GUID of an Information Object and returns an array with its assigned properties.

`setInformationObjectPayload()` takes the GUID of an Information Object along with a pointer to raw content and the later is assigned to the former.

`getInformationObjectPayload()` takes a GUID and returns a pointer to raw content if available.

`getInformationObjectRelationships()` takes the GUID of an Information Object and returns an array of the Relationships it is associated with.

`createRelationship()` creates a new Relationship.

`storeRelationship()` persists the Relationship

`retrieveRelationship()` takes a GUID and returnes a Relationship.

`removeRelationship()` deleted the Relationship with the given GUID for the storage.

`getRelationshipProperties()` takes the GUID of a Relationship and returns an array with its assigned properties.

`setProperty()` takes the GUID of the entity and the property name and assigns the given value to the property

`unsetProperty()` removes a property from the entity with the given GUID

`retrieveProperty()` takes the GUID of an entity and returns the value of the given property.

4.4.2 Data Defintion

A portion of the API is for creating and manipulating types. We call it a *Data Definition Langueage (DDL)*. Not every member of the adapting layer is obliged to conform or support it. On the contrary, some adapter implementations could require that DDL is disabled, either completely, exposing a ``read-only'' model were you instantiate objects from a predefined set of types, or partially, if it is allowed to create types only by extending this set.

`loadTypes()` loads the available system types.

`createType()` creates a new type.

`storeType()` stores a type.

`retrieveType()` gets a type.

`destroyType()` destroys an type.

Chapter 5

Conclusion

5.1 Contribution and Future work

Management of data has never been a tougher challenge. Its volume and complexity might pose great obstacles in securing its preservation and enabling its efficient manipulation. We live in the era where the one with the real advantage is not he who just has access to the information but the one capable of effectively search in it to find what he really wants.

In the introductory chapter of this thesis, we have set two main objectives for this work. One, to define a generic yet sufficiently expressive information model for rich content representation and two, propose the outline of a storage system architecture with the twofold purpose of enabling the homogenisation of diverse storage technologies and elevating the capabilities of storage architectures to reach the rich expressiveness of our proposed information model.

We started pursuing the above goals with a review of the work found in the respective literature, where we observed, that in the last decade, a big change of focus is carried out in the world of storage technologies. From a raw capacity hardware competition, to a more sophisticated software information management systems that are able to handle enriched data representations.

During, this study, we were able to pinpoint the weaknesses and inabilities of the state of the art to support our intended functionality. Thus, we proposed an information model for digital content definition and representation. The model comprise information objects and relationships as well

as property values. The backing of a type system, further refines our model's capabilities and promotes reusability. Types grant the definitions of a class of information objects or relationships, comprised of a set of common specifications we called rules. Every instance of that type then, must automatically conform to the type's specifications such as *i)* contain similar metadata and digital content, *ii)* participate in the same relationships, and *iii)* expose uniform behaviour .

We, then, proposed a reference architecture for storage that uses the aforementioned information model to promote a system that homogenises several models, aggregates diverse storage capacities and capabilities into a common resource pool and administers the distribution of the content along underlying storage. The architecture's most notable feature regarding the physical storage requirements is that it can exploit already installed and configured infrastructures, as they are. This is, obviously, cost effective, but also promotes openness for interoperability with applications and systems outside our ecosystem. The proposed architectures comprises three layers. The adapting layer that is bound to the underlying storage technology it serves and is responsible of mapping all or part of our model using constructs available to the underlying storage. The management layer, exploits the underlying adapting layer, to leverage the homogenisation of the diverse storages available. That way, added value services could be implemented through some brokerage mechanism that could route information along the storage pool, based on factors like cost, storage capabilities or even application restrictions. Last but not least, the global reference point, that implements a generic but powerful API, offering overlying client applications, transparent access to a unified storage resource pool, capable of effective and efficient handling of user-defined, complex information structures.

For the future, there has to be a concrete implementation of the architecture since larger scale application of our information model will, certainly, bring up weaknesses it may have, most probably on resolving constraints for rules in a free of conflict manner. Additional difficulty in that part, might also come from multiple inheritance as well. Another important point is to solve problems of processing speed and browsing the model, which are analogous to problems of RDF processing.

As for the architecture services, we have not mentioned anything about lookup and retrieval. Defining some quering scheme is a tough undertaking since it implicitly concerns other important and sophisticated functionality such as indexing, content and metadata transformation, query optimisation, caching, to name just a few of them.

As far as this thesis work is concerned, the objectives have been met. The fruitful discussion of the work to be done proves nothing else but the fact that we achieved to begin a foundation of research and development for a system that sets a common ground for interoperability of content, storage,

data and information management systems that ultimately expose their content to other services. In this manner, the work been done here can only act as a starting point for further observation and contemplation and feel confident that the potential of the idea is to support both useful technical skills and knowledge as well as produce high quality research matter.

Appendix A

Model Implementation

A.1 Introduction

In Chapter 3 we introduced our information model; a model for defining and representing rich information hierarchies. Here, we elaborate on a first prototype design and implementation of this model. Our intention was to focus on some basic features that scribe a course for an efficient and extensible software design. The implementation, for this reason, is neither thoroughly tested and debugged nor complete feature-wise.

A.2 Code Structure

The code to be discussed in this Appendix, can be found in the optical disc that accompanies this thesis report. It is written in *Java* and tested on a Java Runtime Environment version 1.6 (see Listing A.1), though there should be no problem running on any version from 1.5 or later.

Listing A.1: The JRE used (output of `java -version`)

```
java version "1.6.0_22"
OpenJDK Runtime Environment (IcedTea6 1.10.2) (6b22-1.10.2-0ubuntu1
    11.04.1)
OpenJDK Server VM (build 20.0-b11, mixed mode)
```

The source files come organised in *Java packages* as shown in Listing A.2.

Listing A.2: The `gr.uoa.di.madgik.content.model` package organisation.)

```
gr.uoa.di.madgik.content.model
-- exceptions
-- instance
-- pvalue
-- type
-- util
```

A small description of the `gr.uoa.di.madgik.content.model` package and its underlying hierarchy is summarised in the following.

`gr.uoa.di.madgik.content.model` is the root package. Additionally includes model-wide basic classes.

`gr.uoa.di.madgik.content.model.exceptions` organises the model's exception classes.

`gr.uoa.di.madgik.content.model.instance` holds the classes that represent the model's instantiable entities.

`gr.uoa.di.madgik.content.model.pvalue` includes classes implementing wrappers for values of primitive Java types and their corresponding class equivalents.

`gr.uoa.di.madgik.content.model.type` organises the model's type system implementation classes

`gr.uoa.di.madgik.content.model.util` includes a utility class.

For the reader's convenience the code is extensively documented with both inline comments as well as *Javadoc*. For easy access of the later, there is a generated *Javadoc* HTML hierarchy under the `doc` folder in the optical disc which provides quick, hyperlinked access to class documentation, that may prove useful companion for the rest of this Appendix.

A.3 Types

Let us begin with the types. As already mentioned, a type system enhances our model'As capabilities and promotes reusability, so it is a very important aspect of our information model. What types actually do is grant the definitions of a class of similar entities. What defines these ``minimum similarities'', in our world, is the metadata an information object or relationship can carry, the relationships in which an information object can participate or the types of information objects a relationship can link together.

As you can observe, there is a common part for both information object types and relationship types as well as distinctive portions of what we called ``similarity''. Our first attempt was to find the greatest common denominator for both, in order to extract an interface of behaviour that is mutual to any kind of type. This resulted in a `Type` interface which, for example, defines access methods to a set of allowed properties. What determines, an allowed property is the combination of a property name and property value type (not the actual value). Additionally, any type might have zero, one, or more types from which it inherits. This means the interface should expose additional methods for accessing this array of supertypes as well as manipulating inherited behaviour[1]. Moreover a type can be either final if we want to forbid other types from inheriting from it or abstract if we want to prohibit objects from having it as a type unless it is extended by an other, non-abstract type.

For most of the `Type`'s methods, the implementation is common for both information objects and relationships. For example how to define the set of properties they can carry and how they can access or modify them is done exactly the same way. For this reason we created a class named `TypeDefinition` to implement the `Type` interface. We made it abstract in order *i)* not to allow direct use of this class, and *ii)* to free the class from the commitment of implementing the whole interface .

The later focuses on the common behaviour part and leaves specialized implementation to `TypeDefinition` subclasses. It is important to denote, that subclasses, also implement the `Type` interface. In our types ``realm'', these subclasses are the `InformationObjectType` and `RelationshipType` classes. These, implement information object and relationship specific parts of the `Type` interface or additional behaviour they define on their own. In our implementation, information object types hold a collection of allowed relationships and relationship types hold a

[1] At this point we have implemented a property inheritance mechanism which employs a naive conflict resolving tactic where if a type's property name is found on a parent as well, then the later is ignored whatsoever (which of course can and should be enhanced in subsequent implementation efforts.

collection of allowed source type - target type pairs.

Figure A.1 depicts the inheritance relationships among type classes, which we discussed earlier. Note that even though we make `InformationObjectType`s and `RelationshipType`s extend `TypeDefinition` we make them conform to the `Type` interface in order to preserve the behaviour and later take advantage of subtyping semantics of Java.

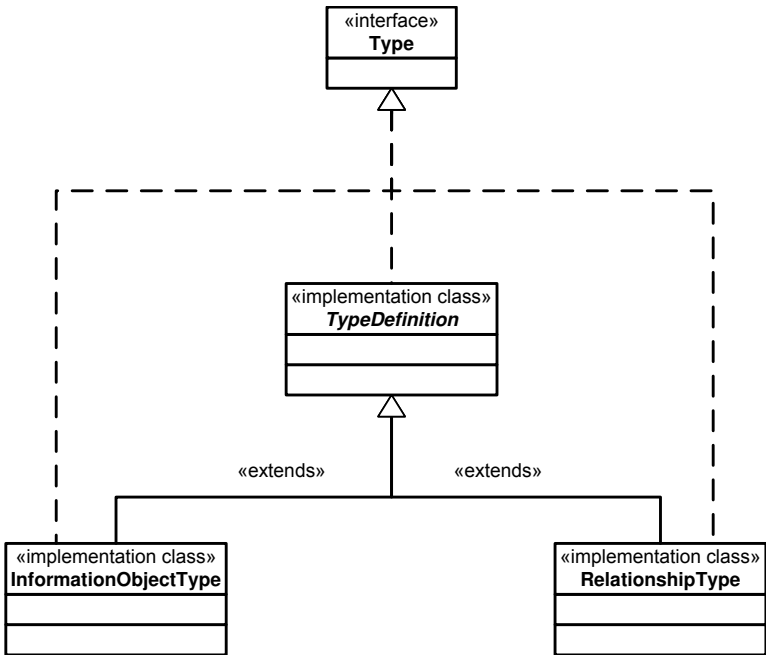


Figure A.1: Inheritance relationships among type classes.

Property value types do not provide any dynamic tools for type manipulation. There is a set of eleven primitive value types which can be used as primitives, without changing them. This special type is realised through an enumeration which is presented in Figure A.2.

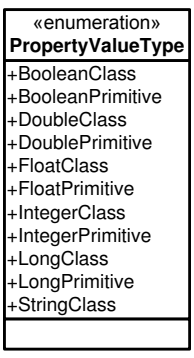


Figure A.2: The property value type implementation.

A.3.1 Type Serialisation

Information object types and relationship types are specifications of the structure and the behaviour of a group of entities, information objects and relationships, respectively. Such specifications can be expressed in XML format. The XML type definitions can then be loaded by the framework at system start up, in order to be later used as type during runtime. The system then is able to treat an individual object as being an instance of its type, automatically complying with that type's specifications.

Example Scenario

For example, consider a paradigm with `Music Albums` that compile `Audio` as songs and `Image` as coverart.:

- `Audio` objects are realised in terms of some property metadata along with two versions of the audio stream; a version containing the full stream and a version with a sample. Moreover, a relationship rule that denotes that it can participate in `compiles` typed relationships.
- `Image` objects similarly contain some properties along with three versions of the containing digital image; a high quality for preserving the image, a web quality for publishing the image on the Web and a thumbnail image for thumbnail usage. Moreover, a relationship rule denotes
- `Music Albums`, do not contain any digital content but hold some metadata and links to `Audio` for the songs and a link to `Image` for coverart.

Listing A.3 contains an example of the definition of an `Audio` type.

Listing A.3: The serialization of an `Audio` IOT

```
<?xml version="1.0" encoding="UTF-8"?>
<iot id="audio" xmlns="http://www.example.org/iot">

    <label>Audio IOT</label>
    <description>This IOT defines the information objects that...</
       description>

    <properties>
        <property name="encoder" valuetype="StringClass"/>
```

```
        <property name="bit-rate" valuetype="IntegerPrimitive"/>
        <property name="sample-rate" valuetype="IntegerPrimitive"/>
    </properties>

    <digitalcontent>
        <stream id="full">
            <label>Full Audio</label>
            <description>The full audio used for...</description>
            <mime type="audio/wav"/>
            <mime type="audio/x-flac"/>
            <mime type="audio/x-mp3"/>
        </stream>
        <stream id="sample">
            <label>Sample Audio</label>
            <description>The sample audio used for...</description>
            <mime type="audio/mpeg"/>
            <mime type="audio/vorbis"/>
            <mime type="audio/x-ms-wma"/>
        </stream>
    </digitalcontent>

</iot>
```

Listing A.4: The serialization of a Song IOT

```
<?xml version="1.0" encoding="UTF-8"?>
<iot id="song" xmlns="http://www.example.org/iot">

    <label>Song IOT</label>
    <description>This IOT defines the information objects that...</
       description>
    <inherits>
        <iot id="audio"/>
    </inherits>

    <properties>
        <property name="title" valuetype="StringClass"/>
    </properties>

    <relationships>
        <relationship type="compiles" rule="allowed"/>
    </relationships>
```

```
</iot>
```

Listing A.5: The serialization of an `Image` IOT

```
<?xml version="1.0" encoding="UTF-8"?>
<iot id="image" xmlns="http://www.example.org/iot">

    <label>Image IOT</label>
    <description>This IOT defines the information objects that...</
       description>

    <properties>
        <property name="device" valuetype="StringClass" mandatory="false"/>
        <property name="date-taken" valuetype="LongPrimitive" mandatory="
           false"/>
    </properties>

    <relationships>
        <relationship type="has-coverart"/>
    </relationships>

    <digitalcontent>
        <stream id="hq">
            <label>Hi Quality Image</label>
            <description>The hi quality imaged used for...</description>
            <mime type="image/tiff"/>
            <mime type="image/jpeg"/>
        </stream>
        <stream id="web">
            <label>Web Quality Image</label>
            <description>The Web quality image for...</description>
            <mime type="image/jpeg"/>
            <mime type="image/gif"/>
        </stream>
        <stream id="thumb">
            <label>Thumbnail Image</label>
            <description>The thumbnail image used for...</description>
            <mime type="image/gif"/>
            <mime type="image/png"/>
        </stream>
    </digitalcontent>
```

```
</iot>
```

Listing A.6: The serialization of a `Music Album` IOT

```
<?xml version="1.0" encoding="UTF-8"?>
<iot id="music-album" xmlns="http://www.example.org/iot">

    <label>Music Album IOT</label>
    <description>This IOT defines the information objects that...</
       description>

    <properties>
        <property name="title" valuetype="StringClass" mandatory="false"/>
        <property name="artist" valuetype="StringClass" mandatory="false"/>
        <property name="composer" valuetype="StringClass" mandatory="false
           "/>
        <property name="genre" valuetype="StringClass" mandatory="false"/>
        <property name="year" valuetype="StringClass" mandatory="false"/>
    </properties>

    <relationships>
        <relationship type="compiles"/>
        <relationship type="has-coverart"
    </relationships>

</iot>
```

Listing A.7: The serialization of a `Compiles` RT

```
<?xml version="1.0" encoding="UTF-8"?>
<rt id="compiles" xmlns="http://www.example.org/rt">

    <label>Compiles RT</label>
    <description>This RT defines the relatioships denoting the compilation
       of of songs by a music album</description>

    <properties>
        <property name="is-single" valuetype="BooleanPrimitive" mandatory="
           true"/>
        <property name="track-no" valuetype="IntegerPrimitive" mandatory="
           true"/>
```

```
    </properties>

    <roles>
        <link source="music-album" target="song" multiplicity="many"/>
    </roles>

</rt>
```

Listing A.8: The serialization of a `Has Coverart` RT

```
<?xml version="1.0" encoding="UTF-8"?>
<rt id="has-coverart" xmlns="http://www.example.org/rt">

    <label>Has Cover Art RT</label>
    <description>This RT defines the relatioships that...</description>

    <roles>
        <link source="music-album" target="image" multiplicity="one"/>
    </roles>

</rt>
```

Such a model for songs, coverart and music albums is not universal. Keep in mind that the power of this dynamic nature of a type system is the fact that allows the definition of arbitrary types of information objects and relationships. For this example the designer generated the above schema to meet the specific requirements at hand. If things change then he is free and above all able to structure the data however fits him best.

A.4 Instances

We call instances, all the classes that are being dynamically realised inside a system that exposes our proposed information model. These are the building blocks of the representation of semantically rich data hierarchies. This ``realm'' comprises information objects, relationships and property values. The simple conceptual class diagram of these notions is shown in Figure A.3.

The basic outline of this static class diagram is that we have `InformationObject` and

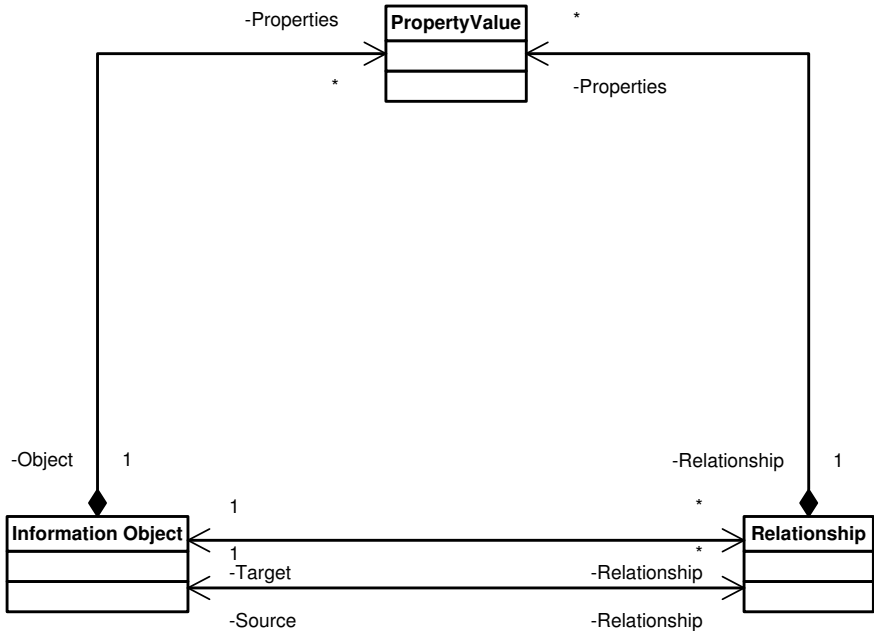


Figure A.3: The basic conceptual class diagram of the information model.

`Relationship` classes. They are both composed by an number (zero, one or more) of property values[2]. `InformationObject`s and `Relationship`s can be associated with links. The former can play the role of either the source or the target of the relationship. Moreover, it can be the source or the target for other relationships too. On the other hand, relationships can have only one source and one target information objects.

Similarly to how we have handled types in our implementation, the instances have a similar class diagram, as it is depicted in Figure A.4.

A generic `Instance` interface provides the common abstract behaviour of all kinds of instances, either information objects or relationships. For example it defines an abstract method, called `getType()`, which every implementing class must implement in order to return its type (see Listing A.9).

Listing A.9: Getter of the type of an instance

```
public abstract Type getType();
```

Also, the interface provides access and manipulation methods for a map of property values. Note that instances have property values. In order to find out which is the type of this property value then

[2]Note that the precise semantic meaning of a property value is being given by its corresponding property value type.

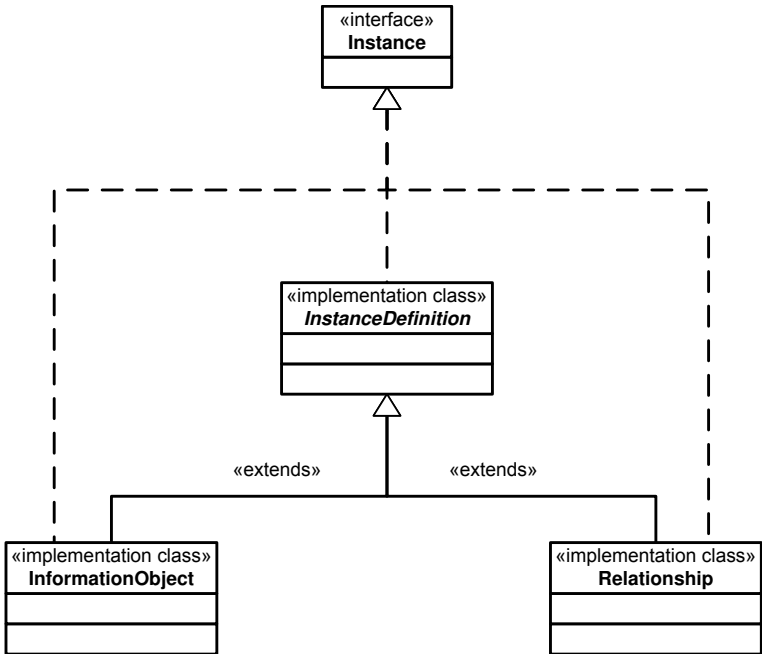


Figure A.4: Inheritance relationships among instance classes.

the property value types of the type of this instance should be traversed to retrieve the corresponding entry.

`InstanceDefinition` is an abstract class implementing part of the `Instance` interface. In our implementation, it sticks only with the property values manipulation. For accessing an instance's type, the implementation of the `getType()` is written in the `InformationObject` and `Relationship` classes, seperately. These classes encapsulate the corresponding type reference which is of course, an (`InformationObjectType` for `InformationObject`s or a `RelationshipType` for `Relationship`s). The code realisation is a great example of the use of Java's covariant return type mechanism where an implementing class can narrow the return type of an abstract method[3] (see Listings A.10 and A.11.

Listing A.10: Narrowing the return type from `Type` to `InformationObjectType`

```
@Override
public InformationObjectType getType() {
    return type;
}
```

[3]Unfortunately, this feature is not applicable for abstract method parameters.

and

Listing A.11: Narrowing the return type from `Type` to `RelationshipType`

```
@Override
public RelationshipType getType() {
    return type;
}
```

Apart from its type, an `InformationObject` contains a collection of the relationships in which it participates and, if available, its binary payload (or maybe a reference to it). `Relationship` specific assets include a reference to the information object which plays the role of the source and a reference to the information object which plays the role of the target.

A more conceptual approach would omit properties as a basic elements of the model, as these are shown in Figure A.3), abstracting them as information objects themselves, associated with other information objects using, for example an `is-property-of`-like relationship. Since our principal target is storage systems, we observed that in filesystems for instance, entities are always accompanied by a number of cardinal storage attributes, represented as simple key-type-value associations. Properties, thus, are not considered objects themselves but a value of some kind of primitive data type (integer, string, boolean e.t.c). A diagram depicting association and inheritance relationships among the `PropertyValue` interface and its corresponding implementing classes, like `BooleanPrimitive`, is shown in Figure A.5.

A.5 Overview

An overview of the discussion of the implementation of our proposed information model is pictured in Figure A.6. It shows both instances and types as well as property values and their inheritance and association relationships. The short story it tells can be summarised as follows. Every instance in the model has exactly one type while a type may have multiple instances. This type can have zero, one or more parent supertypes through its type definition. Additionally a type can be a supertype for many other types. A property value has exactly one property value type, while in the opposite direction, a property value type, many values. Also, qualified associations are realised between `InstanceDefinition`s and `PropertyValue`s as well as between `TypeDefinition`s and `PropertyValueType`s. To be more specific, for a given combination

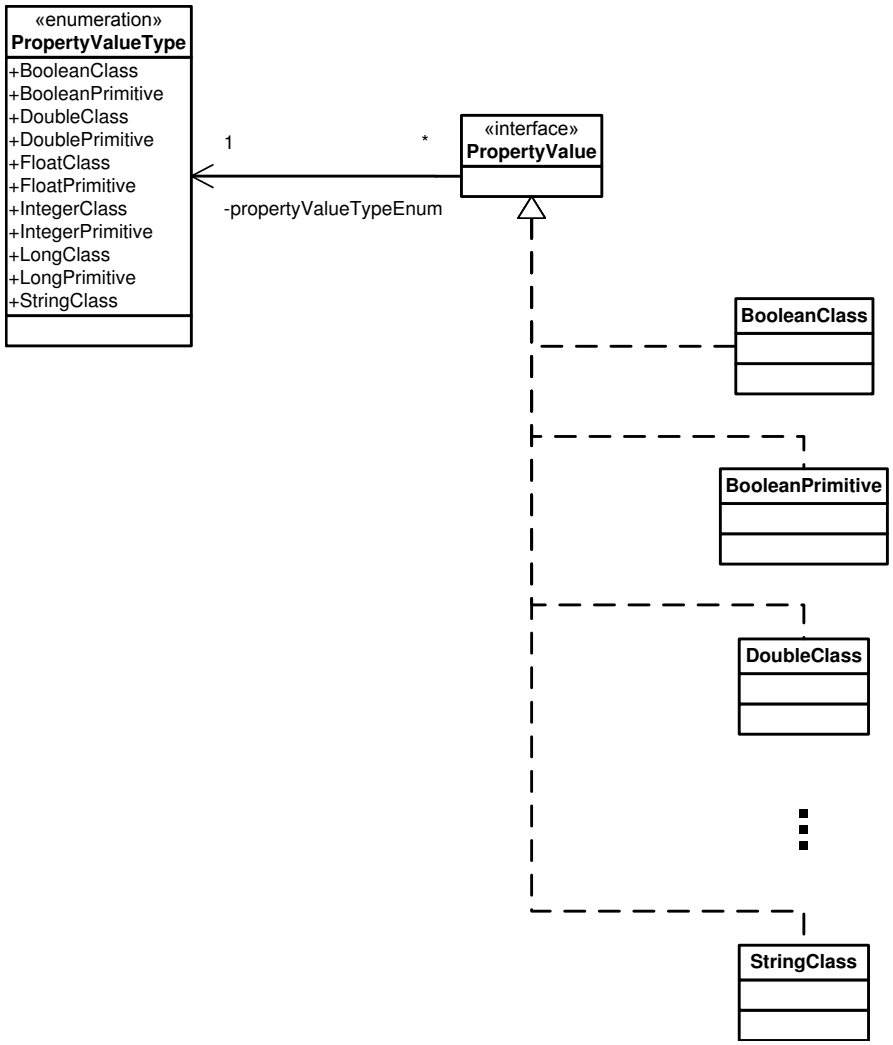


Figure A.5: Association and inheritance relationships for the `PropertyValue` interface.

of `InstanceDefinition` and (property) `name` there can be zero or one `PropertyValue`. Similarly, for a given combination of `TypeDefinition` and (property) `name` there can be zero or one `PropertyValueTypes`. Qualified associations are realised in Java through the use of classes that implement the `Map` interface (e.g. `Hashtable<K, V>`).

Last, you may have noticed in the Java code, that both instances and types extend the `Element` abstract class. This is achieved through the definition classes `InstanceDefinition` and `TypeDefinition`. The static class diagram of Figure A.7 shows the corresponding extension relationships.

An `Element` class is considered a basic, model-wide entity. It is implemented by an abstract

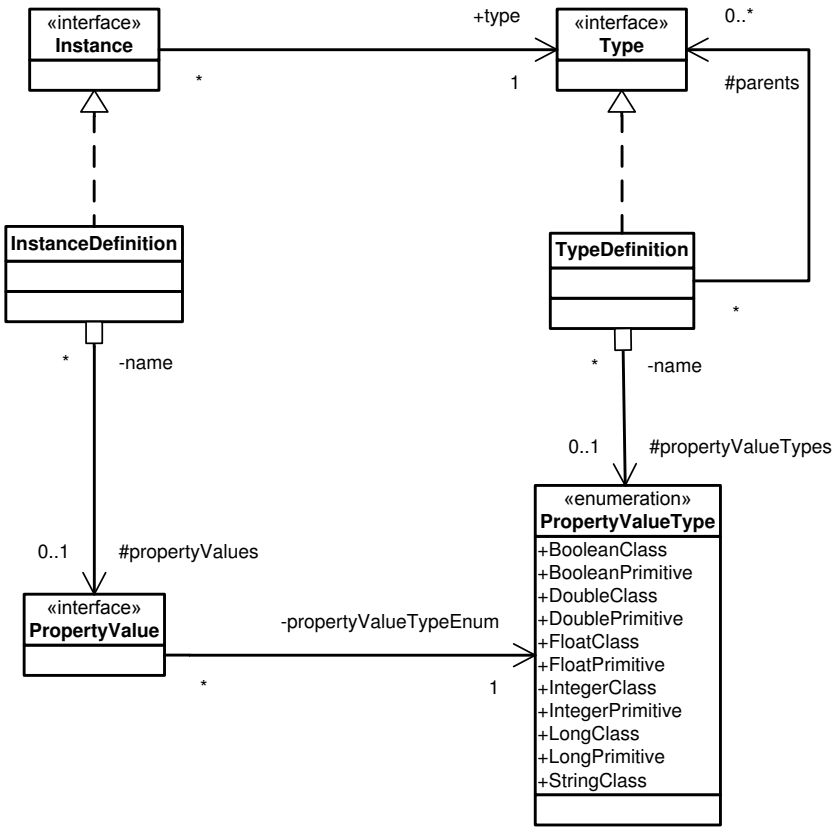


Figure A.6: An overview of the model's entities and their inheritance and association relationships.

class whose role is to attach some basic features to all model citizens[4], namely assign a unique identification to the entity as well as a name and a description.

[4]Except from property value types and property values which, as mentioned before, are considered primitive.

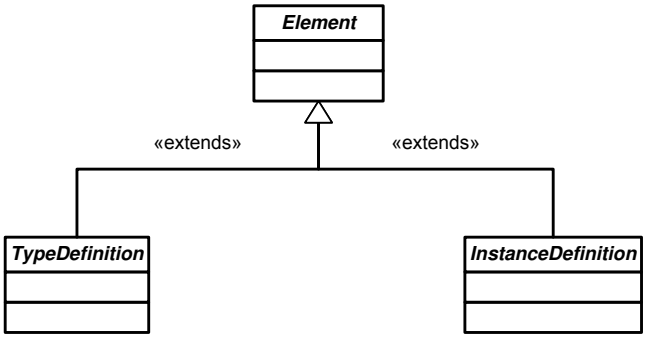


Figure A.7: Both type and instance definitions inherit from `Element`.

Bibliography

(1) Arxiv.org e-print archive. http://arXiv.org. Online; accessed 10-June-2010.

(2) The cnri handle system. http://www.handle.net. Online; accessed 10-June-2010.

(3) Distributed collaboratories infrastructure on grid enabled technology 4 science (d4science). http://cordis.europa.eu/fetch?CALLER=FP7_PROJ_EN&ACTION=D&DOC=1&CAT=PROJ&QUERY=011d29910b73:6ef2:5f230691&RCN=86427. Online; accessed 01-July-2010.

(4) Dspace documentation. https://wiki.duraspace.org/display/DSDOC/DSpace+Documentation. Online; accessed 25-April-2010.

(5) Dublin core library application profile. http://dublincore.org/documents/library-application-profile. Online; accessed 10-June-2010.

(6) Eprints project. http://www.eprints.org. Online; accessed 07-June-2010.

(7) Fedora relationship ontology. http://www.fedora-commons.org/definitions/1/0/fedora-relsext-ontology.rdfs. Online; accessed 08-June-2010.

(8) Greenstone digital library software. http://www.greenstone.org. Online; accessed 08-June-2010.

(9) Metadata encoding and transmission standard. http://www.loc.gov/standards/mets. Online; accessed 12-June-2010.

(10) Open archives initiative - object reuse and exchange. http://www.openarchives.org/ore. Online; accessed 02-July-2010.

(11) Alexander Ames, Carlos Maltzahn, Nikhil Bobb, Ethan L. Miller, Scott A. Brandt, Alisa Neeman, Adam Hiatt, and Deepa Tuteja. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, pages 49--60, Washington, DC, USA, 2005. IEEE Computer Society.

(12) Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. Lifs: An attribute-rich file system for storage class memories. In *In Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies (College Park, MD*. IEEE, 2006.

(13) Sasha Ames, Maya B. Gokhale, and Carlos Maltzahn. Design and implementation of a metadata-rich file system. Technical Report UCSC-SOE-10-07, UCSC, February 2010.

(14) William Y. Arms. Key Concepts in the Architecture of the Digital Library. *D-Lib Magazine*, July 1995.

(15) William Y. Arms, Christophe Blanchi, and Edward A. Overly. An architecture for information in digital libraries. *D-lib Magazine*, 3, 1997.

(16) Christophe Blanchi and Jason Petrone. An architecture for digital object typing. Technical report, CNRI, September 2001.

(17) Christophe Blanchi and Jason Petrone. Distributed Interoperable Metadata Registry. *D-Lib Magazine*, 7(12), 2001.

(18) Jeremey Carroll. Named Graphs. `http://www.w3.org/2004/03/trix`, November 2004. Online; accessed 28-May-2010.

(19) Ccsds. Reference Model for an Open Archival Information System (OAIS) : Recommendation for Space Data System Standards : CCSDS 650.0-B-1. Technical report, January 2002.

(20) J. Clark and S. DeRose. XML Path Language (XPath) version 1.0, W3C Recommendation. `http://www.w3.org/TR/xpath`, November 1999. Online; accessed 30-May-2010.

(21) Quentin Clark. WinFS Update, What's in Store, MSDN Blogs. `http://blogs.msdn.com/b/winfs/archive/2006/06/23/644706.aspx`, June 2006. Online; accessed 02-Feb-2010.

(22) Kati Dimitrova. About WinFS Rules, MSDN Blogs. `http://blogs.msdn.com/b/winfs/archive/2005/10/10/479416.aspx`, October 2005. Online; accessed 09-Feb-2010.

(23) Andrew Fryer. Beyond Relational - Installing SQL Server FileTable. http://blogs.technet.com/b/andrew/archive/2011/08/31/beyond-relational-installing-sql-server-filetable.aspx, August 2011. Online; accessed 31-Aug-2011.

(24) John Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. Technical report, International Data Corporation (IDC) whitepaper, sponsored by EMC, March 2008.

(25) John Gantz and David Reinsel. Extracting value from chaos. Technical report, International Data Corporation (IDC) whitepaper, sponsored by EMC, June 2011.

(26) J. Gregorio and B. de hOra. The atom publishing protocol (ietf rfc). http://tools.ietf.org/html/rfc5023, October 2007. Online; accessed 30-May-2010.

(27) Richard Grimes. Code name winfs: Revolutionary file storage system lets users search and manage files based on content, msdn microsoft. http://msdn.microsoft.com/en-us/magazine/cc164028.aspx, January 2004. Online; accessed 11-July-2010.

(28) I. Jacobs and N. Walsh. Architecture of the World Wide Web. http://www.w3.org/TR/webarch, January 2004. Online; accessed 27-May-2010.

(29) JISC. Supporting digital preservation and asset management in institutions. http://www.jisc.ac.uk/whatwedo/programmes/preservation/assetmanagement.aspx, http://http://www.jisc.ac.uk/media/documents/programmes/preservation/404publicreport_2008.pdf, January 2008. Online; accessed 27-April-2010.

(30) K. J. Jon, D. Bainbridge, and I. H. Witten. The design of greenstone 3: An agent based dynamic digital library. Technical report, Technical report, Department of Computer Science, University of Waikato, Hamilton New Zealand, December 2002. last viewed on 15 March 2010 at http://www.sadl.uleth.ca/greenstone3/gs3design.pdf, December 2002.

(31) Robert Kahn and Robert Wilensky. A framework for distributed digital object services. Corporation for National Research Initiatives,Reston, Va., May 1995.

(32) Robert Kahn and Robert Wilensky. A framework for distributed digital object services. *Int. J. Digit. Libr.*, 6:115--123, April 2006.

(33) Carl Lagoze, Herbert Van de Sompel, Michael L. Nelson, Simeon Warner, Robert Sanderson, and Pete Johnston. Object re-use & exchange: A resource-centric approach. *CoRR*, abs/0804.2273, 2008.

(34) Carl Lagoze, Sandy Payette, Edwin Shin, and Chris Wilper. Fedora: An architecture for complex objects and their relationships. *CoRR*, abs/cs/0501012, 2005.

(35) Carl Lagoze and Herbert Van de Sompel. Compound Information Objects: The OAI-ORE Perspective, May 2007.

(36) Yuri Leontiev, M. Tamer Özsu, and Duane Szafron. On type systems for object-oriented database programming languages. *ACM Comput. Surv.*, 34:409--449, December 2002.

(37) LoC. Library of Congress announces awards of $13.9 million to begin building a network of partners for digital preservation. `http://www.loc.gov/today/pr/2004/04-171.html`, September 2004. Online; accessed 25-April-2010.

(38) MacLeonard. Malware utilising alternate data streams? , auscert web log. `https://www.auscert.org.au/render.html?it=7967`, August 2007. Online; accessed 21-July-2010.

(39) F Manola and E Miller. RDF Primer, W3C Recommendation. `http://www.w3.org/TR/rdf-primer`, February 2004. Online; accessed 30-May-2010.

(40) Eric Miller. An Introduction to the Resource Description Framework. *D-Lib Magazine*, May 1998.

(41) NTFS.com. Winfs overview. `http://www.ntfs.com/winfs_basics.htm`. Online; accessed 09-July-2010.

(42) Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, nov 1999. Previous number = SIDL-WP-1999-0120.

(43) Sandra Payette and Carl Lagoze. Flexible and extensible digital object and repository architecture (fedora). In *Proceedings of the Second European Conference on Research and Advanced Technology for Digital Libraries*, ECDL '98, pages 41--59, London, UK, 1998. Springer-Verlag.

(44) Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

(45) T. Rizzo and S. Grimaldi. Data access and storage developer center: An introduction to winfs opath, msdn microsoft. http://msdn.microsoft.com/en-us/library/aa480689.aspx, October 2004. Online; accessed 10-July-2010.

(46) Thomas Rizzo. Winfs 101: Introducing the new windows file system, msdn microsoft. http://msdn.microsoft.com/en-US/library/aa480687.aspx, March 2004. Online; accessed 10-July-2010.

(47) Paula Rooney. WinFS Still In The Works Despite Missing Vista. http://www.crn.com/news/applications-os/196600671/winfs-still-in-the-works-despite-missing-vista.htm?itc=refresh, November 2006. Online; accessed 04-Aug-2010.

(48) Kostas Saidis, George Pyrounakis, and Mara Nikolaidou. On the effective manipulation of digital objects: A prototype-based instantiation approach. In Andreas Rauber, Stavros Christodoulakis, and A Tjoa, editors, *Research and Advanced Technology for Digital Libraries*, volume 3652 of *Lecture Notes in Computer Science*, pages 13--24. Springer Berlin / Heidelberg, 2005. 10.1007/11551362_2.

(49) Kostas Saidis, George Pyrounakis, Mara Nikolaidou, and Alex Delis. Digital object prototypes: An effective realization of digital object types. In Julio Gonzalo, Costantino Thanos, M. Verdejo, and Rafael Carrasco, editors, *Research and Advanced Technology for Digital Libraries*, volume 4172 of *Lecture Notes in Computer Science*, pages 123--134. Springer Berlin / Heidelberg, 2006. 10.1007/11863878_11.

(50) MacKenzie Smith, Mary R. Barton, Margret Branschofsky, Greg McClellan, Julie Harford Walker, Michael J. Bass, David Stuve, and Robert Tansley. Dspace: An open source dynamic digital repository. *D-Lib Magazine*, 9(1), 2003.

(51) J. F. Sowa. *Conceptual structures: information processing in mind and machine*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

(52) Ed Sponsler and Eric F. Van de Velde. Eprints.org software: a review. California Institute of Technology, May 2001.

(53) Thornton Staples, Ross Wayland, and Sandra Payette. The Fedora Project - An Open-source Digital Object Repository Management System. *D-Lib Magazine*, 9(4), April 2003.

(54) R. Tansley and S Harnad. Eprints.org Software for Creating Institutional and Individual Open Archives. *D-Lib Magazine*, 6(10), October 2000.

(55) Robert Tansley, Mick Bass, David Stuve, Margret Branschofsky, Daniel Chudnov, Greg McClellan, and MacKenzie Smith. The dspace institutional digital repository system: current functionality. In *Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries*, JCDL '03, pages 87--97, Washington, DC, USA, 2003. IEEE Computer Society.

(56) David Tarrant, Ben O'Steen, Tim Brody, Steve Hitchcock, Neil Jefferies, and Les Carr. Using oai-ore to transform digital repositories into interoperable storage and services applications. *The Code4Lib Journal*, 6, March 2009.

(57) USNARA. National archives names two companies to design an electronic archives. `http://www.archives.gov/press/press-releases/2004/nr04-74.html`, August 2004. Online; accessed 25-April-2010.

(58) H. Van de Sompel and C. Lagoze. Interoperability for the discovery, use, and re-use of units of scholarly communication. *CTWatch Quarterly*, 3(3), August 2007.

(59) Duraspate Wiki. Fedora 3.5 documentation. `https://wiki.duraspace.org/x/vwCcAQ`, 2011. (Online; accessed 5-October-2011).

(60) Shawn Wildermuth. A developer's perspective on winfs: Part 1, msdn microsoft. `http://msdn.microsoft.com/en-us/library/ms996622.aspx`, March 2004. Online; accessed 10-July-2010.

(61) Shawn Wildermuth. A developer's perspective on winfs: Part 2, msdn microsoft. `http://msdn.microsoft.com/en-us/library/ms996631.aspx`, July 2004. Online; accessed 10-July-2010.