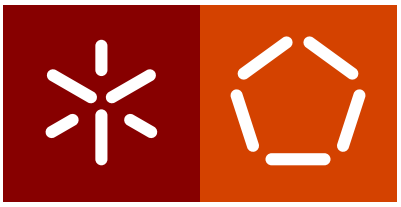
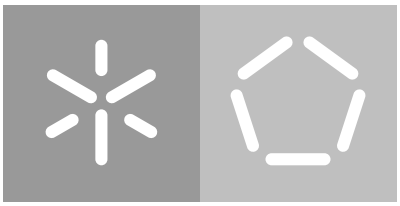


Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Pereira

RODA-in - a generic tool for the mass creation of submission information packages

November 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Pereira

RODA-in - a generic tool for the mass creation of submission information packages

Masters dissertation
Masters in Informatics Engineering

Work under the supervision of
Professor Doctor José Carlos Ramalho
Engineer Luís Faria

November 2016

ACKNOWLEDGEMENTS

This work was only possible due to the people that have always been by my side and helped me in this journey. Here I leave my deepest gratitude for those that have contributed to this thesis.

Firstly, I would like to thank my supervisors which have always been available to help me. To Professor José Carlos Ramalho for believing in me and for the opportunity to work with him. To Luís Faria for all the support and guidance when developing this work, my many thanks.

To KEEP Solutions, the company that welcomed me and provided the resources that enabled the completion of this work. To my colleagues, Miguel Ferreira and Hélder Silva for all the knowledge and experience that they kindly transmitted. And to all my company colleagues, for all their camaraderie.

To DGLAB, which helpfully tested and contributed with feedback to improve this work, namely Francisco Barbedo, Lucília Runa, Mário Santana and Ana Rodrigues. To the E-ARK Project and its members for piloting the application and for their recommendations to enhance this work.

Finally, to my family, especially to my parents, for their enormous efforts on providing me with the best possible education and for always helping me follow my dreams. To my dear Ana Clara for her unceasing encouragement, support and attention, and for being my source of inspiration of hard-work and dedication.

This accomplishment would not be possible without all of them. Thank you.

ABSTRACT

Digital preservation is the sum of activities necessary to ensure the long-term access to digital information. The OAIS standard(ISO, 2012a) was developed in order to ease the communication between the various entities involved in the preservation of digital objects and regulate the long-lasting storage of digital information.

The preservation process begins when the producer creates Submission Information Packages (SIP) and uploads them to the archive's repository. To create these packages, the producer must choose which files to archive and provide extra information (metadata) to describe and to allow finding the information. As the production of digital content increases exponentially, the creation of SIP by current methods can be too onerous and even unfeasible.

This work focuses on creating a semi-automatic way of producing SIPs by employing a simple and well-defined workflow. Using the file system as the source of content, the producer defines aggregation and metadata association rules and specify how the SIPs are created. The application that was developed to support this work, RODA-in, was designed to be able to create thousands of SIPs with gigabytes of data in an easy to use way. Additionally, it has multiple features that ease the work of the producer, such as metadata templating and mass edition.

RESUMO

A preservação digital define-se pelo conjunto das atividades necessárias para garantir o acesso a longo prazo à informação digital. A norma OAIS(ISO, 2012a) foi desenvolvida para facilitar a comunicação entre as várias entidades envolvidas na preservação de objetos digitais e regular o armazenamento duradouro da informação digital.

O processo de preservação começa quando o produtor cria Pacotes de Informação de Submissão (SIP) e os envia para um repositório de um arquivo. Para criar estes pacotes, o produtor tem que escolher os ficheiros que pretende arquivar e fornecer informação extra (metadados) para descrever e permitir a descoberta da informação. Uma vez que a produção de conteúdo digital tem vindo a crescer exponencialmente, a criação de SIP pelos métodos atuais pode ser demasiado oneroso ou mesmo impraticável.

Este trabalho foca-se na criação de uma forma semi-automática the produzir SIPs empregando um *workflow* simples e bem definido. Usando o sistema de ficheiros como fonte do conteúdo, o Produtor define regras de agregação e de associação de metadados que especificam como os SIPs são criados. A aplicação que foi desenvolvida para suportar este trabalho, RODA-in, foi desenhada para ser capaz de criar milhares de SIPs com gigabytes de dados e para ser de fácil utilização. Adicionalmente, possui múltiplas funcionalidades que facilitam o trabalho do Produtor, como por exemplo a criação de metadados a partir de um modelo ou a edição em massa de metadados.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

This chapter presents the motivation for this dissertation and the objectives that must be accomplished. Furthermore, it will also describe the structure of the remaining document.

1.1 MOTIVATION

The latest developments in digital technology encouraged companies and institutions to use and create data in a digital format. In addition to the digitally-born data, a large portion of previously created information is being digitalized in order to facilitate its access and preservation. However, current processes for archiving and preserving information are not ready to cope with the scale of production of digital information.

Digital preservation was developed to counter this problematic, which consists in the ability of guaranteeing that the digital information remains accessible and with enough authenticity qualities, properties and attributes that it can be interpreted in the future, using a different technological platform from the one used in the moment of its creation. (Ferreira, 2006) Preservation is a continuous process, it aims to detect threats and act on them to mitigate the problems.

The advent of the digital revolution has introduced a profound change in the way archives operate, and the well-established processes of transferring content from producers to archives do not cope well with the specifics of digital records. New methods and tools must be developed to facilitate the processing and archival of the large amount of data currently used.

Although concepts in this area are already matured, e.g. the concepts of submission information packages (SIP) and ingest processes are defined on the Reference Model for an Open Archival Information System (CCSDS, 2012), there has been a lacking of open source tools to support the Producers in the creation of such packages. One common approach is the development of specific integrations between Electronic Records Management Systems (ERMS) in use at the producers side and the archive, which allow the creation of SIPs directly from these systems and submission to the archive's ingest workflow. However, these integrations focus only on the usual suspects, failing when more heterogeneous systems are in place or when producers use niche systems to support their records management

activities. Furthermore, producers may not use a document management system at all and simply manage their records on a shared folder on the local network.

When systems integration is not possible due to lack of support, expertise or budget, the alternative is to create the SIPs manually. While many archival systems provide tools to help with the creation of SIPs, e.g. Web forms or downloadable software, tools that are designed to create SIPs one-by-one are highly inefficient and even unusable in contexts where a producer has a high volume of information that needs to be sent to the archive. An example of one of these tools is *Exactly* from AVPreserve[1] which enables users to create packages and send them to the archive by FTP[2]. Each package must be created individually by first selecting the source files and then choosing the metadata that will be associated to the source material.

1.2 OBJECTIVES

This work describes a novel approach to deal with the scenario where a producer has a high volume of information managed directly on the file system and needs to send it to an OAIS archive.

The main objective of this work is the development of a tool to help producers create packages massively and in an easy way, which will allow them to transfer large amounts of information to the archive's repository. Since the information will be structured in multiple different ways, depending on the producer, the tool must support several granularity levels for creating packages. Generally, the packages contain the material that will be archived and other information that describe the material, called metadata. As such, the tool must support the association between the material and the metadata, both when it already exists and when it still needs to be created. Finally, the tool must be able to create packages in an organized manner, following a plan of classification created by the producer.

1.3 DOCUMENT STRUCTURE

This document is organised in 6 chapters.

The first chapter describes the motivation behind this work, as well as its goals. Its main purpose is to identify the problem at hand and set up objectives that should be accomplished. The second chapter introduces the main concepts of digital preservation, explaining why it is important and what impact it has in the world. It also describes the current state of the art, focusing on tools and what metadata formats are currently used.

1 https://www.avpreserve.com/tools/exactly/
2 FTP means File Transfer Protocol

The fourth chapter explains the proposed approach, describing the RODA-in application. It starts by characterizing the workflow developed and analysing each step individually. Furthermore, it also details all the features that producers may use to extend the basic workflow and tailor the SIP creation to their needs. Finally, it outlines the technologies and major development points of the application, what choices were made, why and how they were implemented.

The fifth chapter provides an evaluation by specialists of the approach described in this work. Finally, the document ends with a description of the conclusions that can be withdrawn from this work, and some future work.

2

STATE OF THE ART

This chapter will present the current technologies and standards of the digital preservation domain, mainly a description of digital preservation, digital object and digital preservation processes. Additionally, it will detail the most common and current metadata formats used in digital preservation as well as the RODA project and its products.

2.1 DIGITAL PRESERVATION

Knowledge preservation has always been a concern and the first records of preservation are arguably attributed to rock paintings of approximately 40 000 years ago. These efforts of our ancestors make up one of the main foundations of history, the artefacts, since without them it would be difficult to describe past events.

Today, a large portion of the intellectual property is created with the help of digital tools. The simplicity by which digital information can be created and shared through modern day communication outlets is a determining factor in the adoption of these tools.

However, even though a digital file can be cloned countless times without loosing quality, it can only be interpreted and consumed with the technological context it's in. Since the world of technology is constantly evolving, the digital data is vulnerable to the rapid obsolescence of most technologies.

An example of digital/technological obsolescence is the BBC Domesday Project, which marked the 900th anniversary of the 11th century census of England, Domesday Book. This new edition of Domesday was published in 1986 and it included a new survey of the United Kingdom, in which people wrote about geography, history, social issues or just about their daily lives. The project was stored in adapted laserdiscs in the LV-ROM format [1], which contained 300 MB of storage space on each side of the disc. The format supported not only analogue video and still pictures, but also digital data. Viewing the information stored in the discs required special hardware: an Acorn BBC Master home computer expanded with a SCSI controller and a specially produced laserdisc player, the Philips VP415 "Domesday Player". In addition, the software was written in BCPL [2], an imperative programming

1 LaserVision Read Only Memory, developed by Philips Electronics
2 Basic Combined Programming Language

language in which C was based on, that never reached the popularity its early promises suggested it might.

As the number of computers capable of reading the format and drives capable of accessing the discs became scarce, there were great fears that the discs would become unreadable. Additionally, emulating the code was very difficult since BCPL was no longer in common use. Fortunately, BBC was able to develop a system capable of accessing the discs using emulation techniques, and extract its contents to a more popular format. The Domesday Project data is now available at the BBC website. [3]

It's important to note that technological obsolescence isn't only found at the physical support level. All of the digital information has to necessarily respect the logical rules of a format. This allows software applications to open and process the stored information adequately. As software evolves, the formats it supports are also a target of updates. It's quite common to find software applications able to load files exported by a previous version of the same application, although this ability rarely goes beyond the last two preceding versions. (Kenney et al., 2003)

The increase of the amount of digital data, created since the proliferation of personal computers, not only in the general population but also in the business world, eventually resulted on the need to create processes and tools to preserve that information. This methodology is different from the one used to preserve a paper document, for example, because it is necessary to ensure that structure, formatting and logic of the digital information is kept throughout the entire process of preservation.

These processes are part of **digital preservation**, which consists in the ability to ensure that the digital information remains accessible and with sufficient quality so that it can be interpreted in the future using a different technological platform from the one used at the time of creation. (Ferreira, 2006) Even though the digital preservation research field has decades of work and focuses on common problems of the present world, the term is not widely known by the masses.

2.1.1 *Digital object*

The basic unit of digital preservation is the **digital object**, which can be defined as any object of information that can be represented using a sequence of binary digits (Ferreira, 2006) (Yamaoka and Gauthier, 2013), for example: databases, videos, audio, digital photographies, documents, presentations and software applications.

The digital object has an ambiguous definition because it can refer to the conceptual aspects and to the intellectual content, formatting and digital structure. Thus, (Thibodeau,

3 http://www.bbc.co.uk/history/domesday

2002) proposed the definition of three new terms that specified the digital object according to the context in which it's expressed.

- The **physical object** is a digital object represented as bits (or symbols) in a medium. Basically, the physical level deals with physical files that are identified and managed by some storage system. At the level of physical storage, the computer system does not know what the bits mean, that is, whether they comprise a natural language document, a photograph, or anything else. Physical inscription does not entail morphology, syntax, or semantics.
- A **logical object** is a digital object processed by software, which gives a meaning to the bits read from the physical mean and creates rules, data structures and formatting data. Although at the storage level the interpretation of the bits is not defined, at the logical level the grammar is independent of physical inscription.
- The **conceptual object** is the digital object recognized by a computer or by a human and it can have differences from the logical object. This is the object we deal with in the real world, it's an entity we would recognize as a meaningful unit of information such as a document, a map, a movie or a song. A conceptual object can be represented in several different digital encodings, for example, a photography taken by a digital camera, can be described in the formats TIFF or PNG, without the loss of any information. This characteristic is very important for the preservation of digital objects.

Figure 1 exhibits several components that structure a digital object, relating them to the three levels defined above: (Faria, 2015)

INTELLECTUAL ENTITY

A set of content that is considered a single intellectual unit for purposes of description and management: for example, a movie, database or particular song. An intellectual entity may include other intellectual entities; for example, a Web site can include a Web page. (PREMIS Editorial Committee, 2015)

DESCRIPTIVE METADATA

Provides information about the intellectual or artistic content of an object. Descriptive metadata is also used for the discovery and identification of the content.

REPRESENTATION

The collection of files needed for a complete rendition of an Intellectual Entity. For example, a website may consist of a single HTML [4] file, in which case the representa-

4 HTML means Hypertext Markup Language

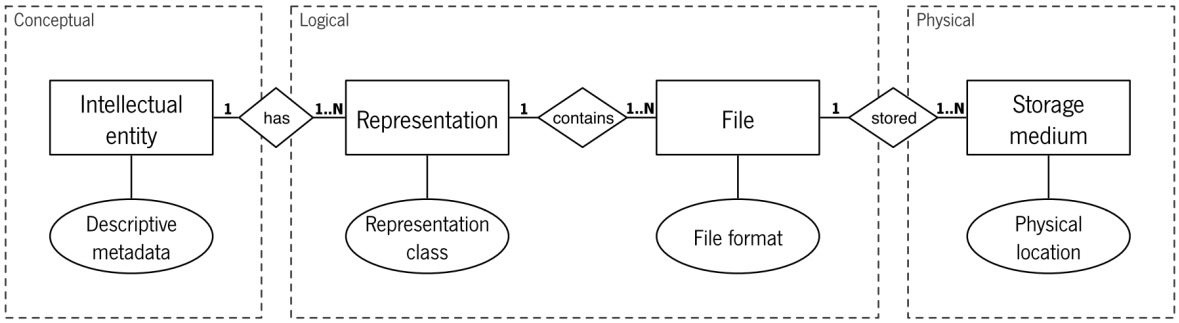


Figure 1: Digital object structure. (Faria, 2015)

tion is composed of that single file, but it can also consist of an HTML file, 2 PNG [5] images and one CSS [6] file, in that case the four files constitute a representation.

REPRESENTATION CLASS

A grouping of representations based on technical properties for digital preservation purposes, for example, web pages.

FILE

A named and ordered sequence of bytes that is known by an operating system. A file can be zero or more bytes and has a file format, access permissions, and file system characteristics such as size and last modification date. (PREMIS Editorial Committee, 2015)

FILE FORMAT

A collection of syntactic and semantic rules used when mapping an information model and a serialized bit stream.

STORAGE MEDIUM

The physical medium on which the object, represented by its files, is stored (e.g. magnetic tape, hard disk, DVD) (PREMIS Editorial Committee, 2015)

PHYSICAL LOCATION

The particular place, position or address of the storage medium.

2.1.2 *Digital preservation processes*

To ease the communication between the several entities involved in the preservation of digital objects, the Consultative Committee for Space Data Systems (CCSDS) in partner-

5 PNG means Portable Network Graphics
6 CSS means Cascading Style Sheets

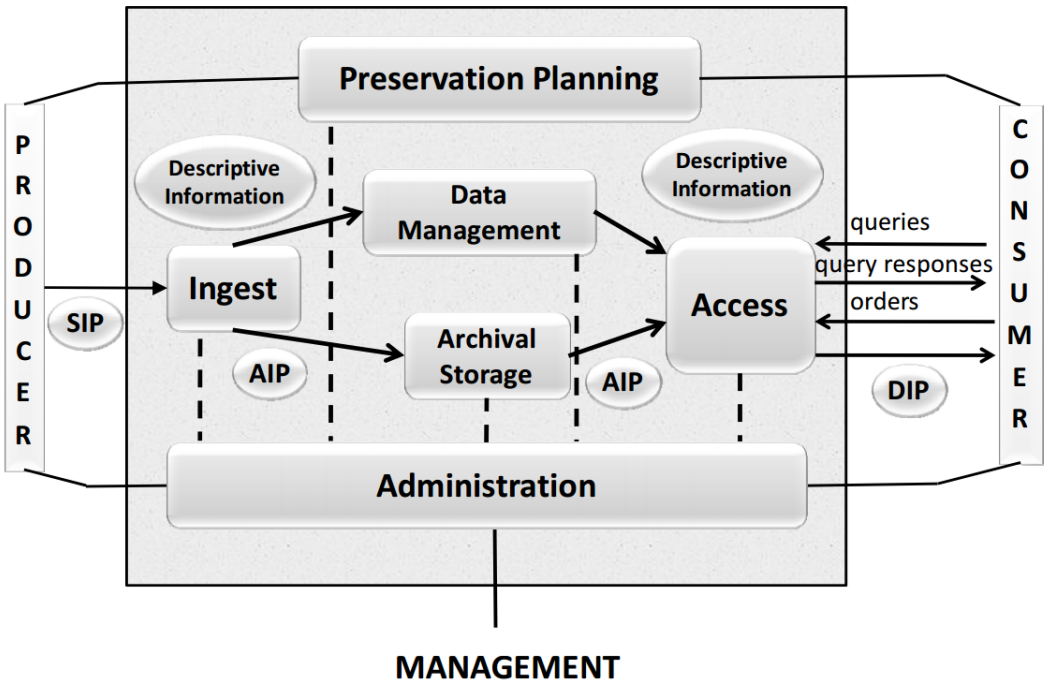


Figure 2: OAIS reference model. (CCSDS, 2012)

ship with the International Organization for Standardization (ISO) defined a set of norms capable of regulating the long lasting storage of digital information, denominated OAIS.

The Open Archival Information System (OAIS) (CCSDS, 2012) is a conceptual model with the objective of identifying the functional components that define an information system dedicated to digital preservation, the objects that are manipulated in its interior and internal and external interfaces. Figure 2 describes a digital repository compatible with the OAIS reference model.

The content creation for archive submission falls under **producer** responsibility. This content is represented in the figure as a **Submission Information Package** (SIP). It's on the producer and in the creation of these submission packets that this dissertation is focused on.

During the submission process, designated as **Ingest**, the system is responsible for verifying the physical, logical an semantical integrity of the data received. At this stage, is validated all the **Descriptive Information** that accompanies the digital representation. This information will support the discovery and localization of the archived material. Should this information fail to be delivered by the producer, it must be created by the system. It's also at the Ingest stage that the **Archival Information Packages** (AIP) are created. The system applies all the necessary operations to transform the SIP in an AIP, i.e., in a logi-

cal structure that is able to unify all the digital representation's components. This is the structure that will be the target of preservation by the OAIS system.

The Descriptive Information, most commonly known as metadata, can be provided by the producer or generated within the system. The **Data Management** component is used to store and manage this information. This unit should, in addition to preserve the descriptive information, allow the establishment of relations between the descriptive metadata and the preserved data, i.e. AIP, ensure the localization of the material and give reports and statistics about the contents of the repository.

The definition of preservation policies and the formulation of preservation plans that act on the stored material are responsibility of the **Preservation Planning** component. This functional unit maintains the data accessible and in accordance with the quality and authenticity requisites defined by the system administrator. It also monitors the environment external to the repository so that it can detect relevant changes of the technological panorama or of the users' requisites that can modify the way the digital information is preserved or disseminated.

The **Access** functional unit establishes the bridge between the system and its community of interest, i.e. the set of potential **Consumers** of the stored content. This unit is responsible for the discovery and dissemination of digital representations, delivering it to the consumer using **Dissemination Information Packages** (DIP). The information delivered to the consumer may be a subset or a transformed version of the archived data, meaning that the DIP are not necessarily equal to the AIP. (CCSDS, 2012) Similarly, since the SIP are generally transformed in data structures that are easier to preserve (although the original object is also preserved), the AIP may be different from the information originally submitted.

Lastly, the **Administration** functional unit interacts with all the other components to ensure the correct operation of the system. Its main activities are monitoring processes, daily log of activities, system parametrization, etc.

Pre-ingest

Many archives (particularly archives with large amounts of data or complex born-digital collections) regularly receive material from producers that contain an array of file formats, project versions and various production material. This material will have to be checked on various aspects, so that a decision can be made on whether it should be accepted to enter the repository or not.

Negotiation phase is the first step. It consists in the definition of pre-conditions, terms and requirements for the content and accompanying information (such as documentation, metadata, contracts, etc.) that will be sent to the repository. An important part of this stage is the creation of a base classification scheme. A classification scheme (or plan) is a list of

hierarchical collections which organize the information according to semantic or logistical aspects, on which the producer may deposit new informational items.

The producer and Repository may sign a written agreement that specifies the type of content and all the legal and technical requirements that both parties are expected to comply, called a **Submission Agreement**.

The next step is to define what information is needed about the records that are about to be archived, such as metadata, file formats, access objectives, etc. This is followed by the conduction of feasibility studies and both the SIP and scope definitions. The process ends with the digital objects ready for submission to the archive, complete with representations, material and descriptive metadata.

There are no universally acknowledged tools or workflows devoted to addressing this phase. The OAIS model supposes that the ingest process starts with SIPs, neglecting how they are created. This phase is not only the one that archives often spend significant time completing, but it's also a critical part of the archiving process. The producer-Archive Interface Specification (CCSDS, 2014) is another CCSDS standard which addresses in more detail the processes around transferring a SIP to the archive, but since it's such a critical and time consuming task, it is still important that this step becomes addressed in the OAIS standard.

2.1.3 *Metadata for digital preservation*

While the OAIS reference model defines a framework with a common vocabulary and provides a functional and information model for digital preservation, it does not detail which specific metadata should be collected or how it should be structured and implemented. The metadata is essential for finding information, providing context and registering dangers and actions, among others. Generally, the metadata for long-term preservation falls into four categories: descriptive, structural, technical and administrative. (Lavoie and Gartner, 2013)

Descriptive metadata is used to describe and identify intellectual entities, collections and related information resources. It uses properties such as the author, title or creation date. It also supports discovery and delivery of digital content and provides an historical context, by, for example, specifying which print-based material was the original source for a digital derivative.

Structural metadata contains information on how the digital object has been created and is important to be able to render the digital object authentically in the future. This type of metadata should be used when a digital object is divisible into component parts and the components are useful themselves. It captures physical structural relationships, such as which pages are part of a book, as well as logical structural relationships, such as

which page follows which in a book. One of its main uses is to facilitate navigation and presentation of complex items by defining structural characteristics such as pagination and sequence.

Technical metadata includes information about the software and hardware on which the digital object can be rendered or executed, checksums and digital signatures to ensure fixity and authenticity. File formats, resolutions, color profiles, compression rations, encryption keys, passwords, brand and model of recording/digitalization equipment, image width and height, are all examples of information that should be added as technical metadata.

Administrative metadata provides information that allows the repository to manage and track objects. It records data about how, when and by whom a resource was created, what preservation actions have been performed on it, and how it can be accessed. It will capture the location of access copies, archival masters and their formats, usually in the form of identifiers, and keep track of the ways a given object is being disseminated, used and/or associated with other digital content. (University of Illinois, 2010) Another important component of administrative metadata is rights and permission management, which may include information about copyright, access, use, licensing and even which preservation actions are permissible. This type of metadata can usually be divided in four sections:

PROVENANCE
: Information that describes the history of an object, tracking all the actions taken during the preservation process. It also records all consequences of the actions which alter the content, appearance or functionality of the object.

RIGHTS
: Data which describes the intellectual property rights that must be respected during the preservation processes or when accessing the object.

SOURCE
: Intended to ensure that the object is really what it should be, i.e., its integrity hasn't been breached in an undocumented way.

TECHNICAL
: Data that describe the technical requisites needed to access, view and use the object. It's under this section that should be aggregated the information about the format of the object, as well as the software, hardware and operating system required to make the object usable, considering the current state of the object in the repository.

Many general and domain-specific metadata standards have been proposed and developed by various user communities. Even within the same subject domain or for the same type of resource, there are often two or more options of metadata standards. (Chan and Zeng)

In Portugal, the more common metadata standards for descriptive metadata are EAD (Enconded Archival Description), Dublin Core, ISAD(G) (General International Standard Archival Description) and CIDOC CRM (Conceptual Reference Model).

2.2 RODA PROJECT

The RODA project [7], started in 2006 and was co-funded by DGARQ (Direção Geral de Arquivos). Its development was possible due to a multidisciplinary team composed of elements of DGARQ and DI/UM. [8]

2.2.1 *RODA*

The main result of this partnership was a software product, named RODA which is still an international reference as a digital preservation solution. RODA is also open-source and is currently under major changes and improvements in order to ease its adoption by public and private institutions. The release of RODA version 2 will be between the end of 2016 and start of 2017.

Of all its characteristics the following can be highlighted:

1. RODA is a digital repository which incorporates all the main functional units the OAIS reference model. (CCSDS, 2012)
2. RODA is able to incorporate, manage and provide access to multiple types of digital material produced within the activities of large companies and public organizations.
3. RODA is built on open-source technologies and is supported by the following norms OAIS, EAD, METS and PREMIS.
4. RODA implements a configurable ingest workflow, which not only validates the deposited packages, but also allows the evaluation and selection of material by the archival personnel.
5. RODA allows the recovery of information in multiple ways - search, classification plan navigation, presentation of representations using specialized viewers and file download.
6. The administration module allows the repository managers to edit descriptive metadata, trigger preservation actions (e.g. integrity checks, format migrations, among

7 Repositório de Objectos Digitais Autênticos or in English: Repository for Authentic Digital Objects http://roda-community.org/

8 Departamento de Informática da Universidade do Minho, or in English: Informatics Department from University of Minho

others), control the accesses by users, view statistics and access logs, among many other options.

7. All of the actions performed in the repository are automatically logged for safety reasons or accountability. RODA is also compatible with ISO 16363:2012 (ISO, 2012b).

2.2.2 *RODA-In*

RODA-in is another product that originated from the RODA project. The application intended to produce submission packages for the RODA digital archive. The packages are created offline, using the producer's computer, and then sent to the archive.

When a new package is being created, the user must fill a few mandatory fields from the EAD metadata schema, for example the title, description level, reference and producer. In addition to these mandatory fields, EAD provides optional fields which range from historical context descriptions to associated materials and even descriptions of content and looks.

Once the descriptive metadata is complete, the producer has to add representations to the package. The supported representation classes are:

TEXT WITH STRUCTURE

Text documents such as PDF [9], Microsoft Word or RTF [10].

IMAGES WITH STRUCTURE

One or more images that may be structured in an hierarchy. These images can be JPEG [11], GIF [12], PNG [13], BMP [14], etc.

AUDIO

Audio files such as MP3 [15] or WAV [16]

VIDEO

Video files such as MPEG [17], AVI [18] or WMV [19]

9 PDF means Portable Document Format
10 RTF means Rich Text Format
11 JPEG is the Joint Photographic Experts Group format
12 GIF means Graphics Interchange Format
13 PNG means Portable Network Graphics
14 BMP are Windows Bitmap image files
15 MPEG-1/2 Audio Layer 3
16 WAV means Waveform Audio File Format
17 MPEG means Moving Picture Experts Group format
18 AVI means Audio Video Interleave
19 WMV is the Windows Media Video format

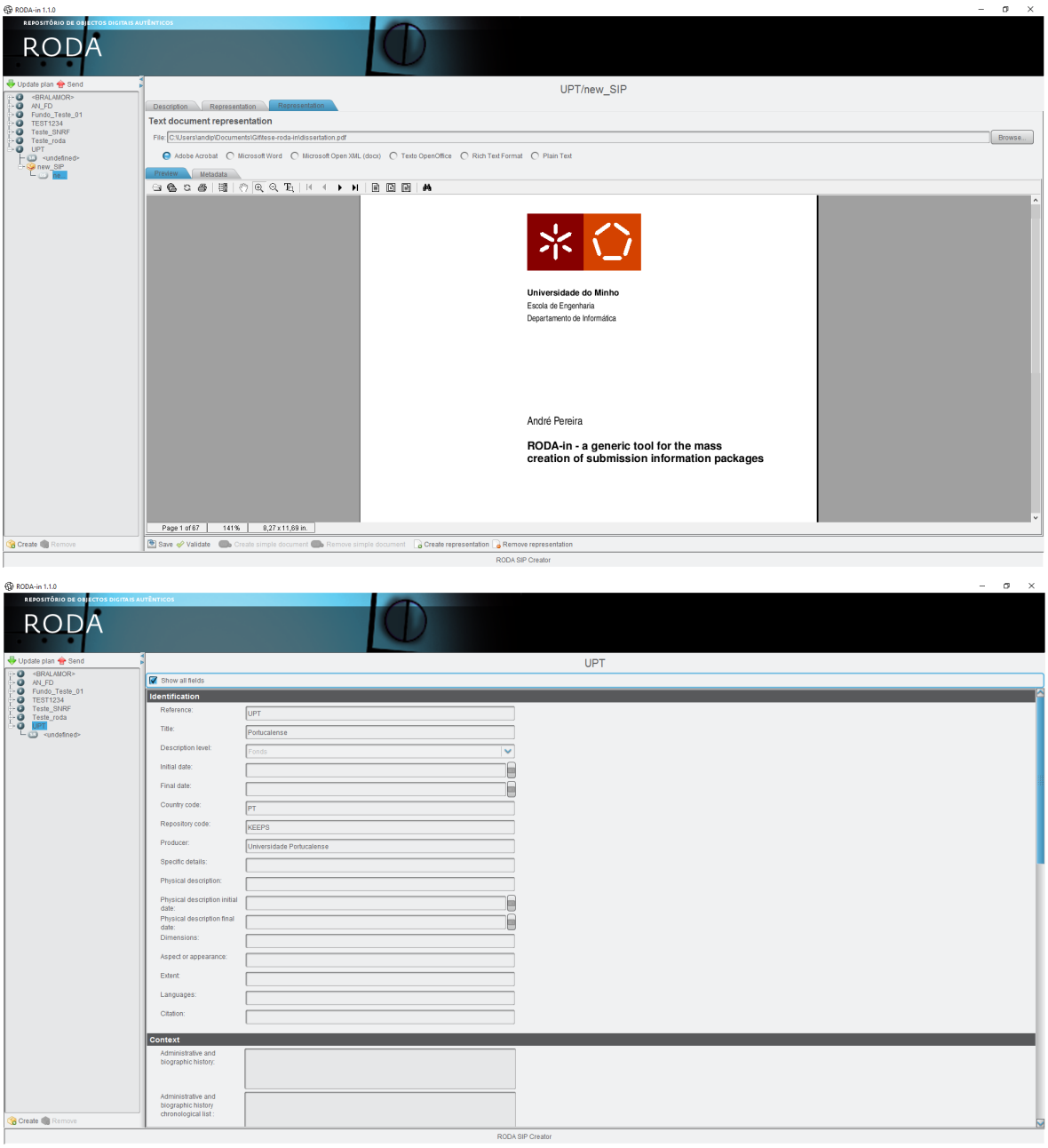


Figure 3: Two screenshots of RODA-in version 1.

RELATIONAL DATABASES

Databases such as MySQL, PostgreSQL, Microsoft SQL Server, Microsoft Access, or even in DBML [20].

UNKNOWN

A document that does not fit in any other option. When using this representation class, the preservation is not guaranteed.

After choosing the class, the application provides a panel for the creation of the representation, where the producer can choose which files to add and its structure. Furthermore, the user may choose to see a preview of the added documents. After all the representations have been added to the package, the user can validate it and send it to the archive.

Figure 3 contains two views of the first version of RODA-in. The left view shows the metadata fields available to the user such as the title, description level, language or administrative and biographic history. The right view displays a preview of one PDF file that was added as a representation to the package. The application opens the file in a PDF viewer since the selected format is "Adobe Acrobat".

Since RODA-in was developed closely with RODA, it's only export option is to send the packages directly to RODA, making it useless if the archive uses other repositories/repository systems.

The creation process described in this section has to be repeated for every submission packed, which is a slow process when the amount of files to be archived is large. The goal of this dissertation is to analyse the most common cases of SIP creation and develop strategies which enable the producer, in few clicks, to produce valid and complete packages *en masse*.

20 DBML means DataBase Markup Language

3

METHODOLOGY

This chapter characterizes the requirements that should be met in this work and explains the approach taken to reach and validate a solution.

3.1 REQUIREMENTS

While concepts in the digital preservation field are already matured, there is still a need for open source tools to help producers with the creation of Submission Information Packages (SIP). The arrival of the digital revolution changed how archives operate, and took the well-established processes of transferring content from producers to archives to the limit. To facilitate the processing and archival of large amounts of digital data, new methods and tools must be created.

Hence, these new methods must follow the following requirements:

1. The user must be able to add, delete and review computer files in the SIP.
2. The tool must allow the user to edit the structural relationships between content files. It must be possible to edit relations among units of description and related computer files / (sub)folders.
3. It must be possible to import previously prepared metadata into the SIP.
4. The tool must support the use of multiple sets of metadata elements and schemas
5. The SIP creation tool must allow manual editing of structured metadata
6. It must allow the creation/edition of classification scheme
7. The tool must produced SIP that can be ingested by any OAIS compliant repository.
8. The solution must be scalable and support multiple levels of granularity when creating SIP.

3.2 APPROACH

The approach chosen was the development of a prototype for an offline desktop tool named RODA-in 2.0. The application must follow all of the previously listed requirements which will be met gradually and validated periodically with its end users. To verify that the requirements are met and to validate the solution, specialists will be consulted and their testimonies presented in chapter 5.

4

RODA-IN 2.0

This chapter will start by describing the workflow and features of the application. Additionally, it outlines the technologies and decisions used in the development of RODA-in and the main aspects of the implementation of major features.

4.1 WORKFLOW

The approach taken was to create an offline application for producers and archivists to create SIPs ready to be submitted to an Open Archival Information System (OAIS). (CCSDS, 2012) The file system works as the source of information of a semi-automated SIP creation process based on rules and heuristics.

The workflow can be described in five steps, as seen in the diagram of figure 4.

1. Choose the folder that holds the data to be archived
2. Import or create a new classification scheme
3. Create a rule to transform the input data to SIPs
   a) Choose SIP-creation aggregation rule
   b) Choose the descriptive metadata association rule
4. Manually edit the created SIPs (Optional)
5. Export the SIPs in the desired format

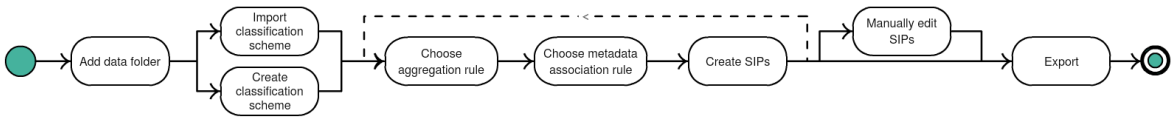


Figure 4: The basic workflow of RODA-in.

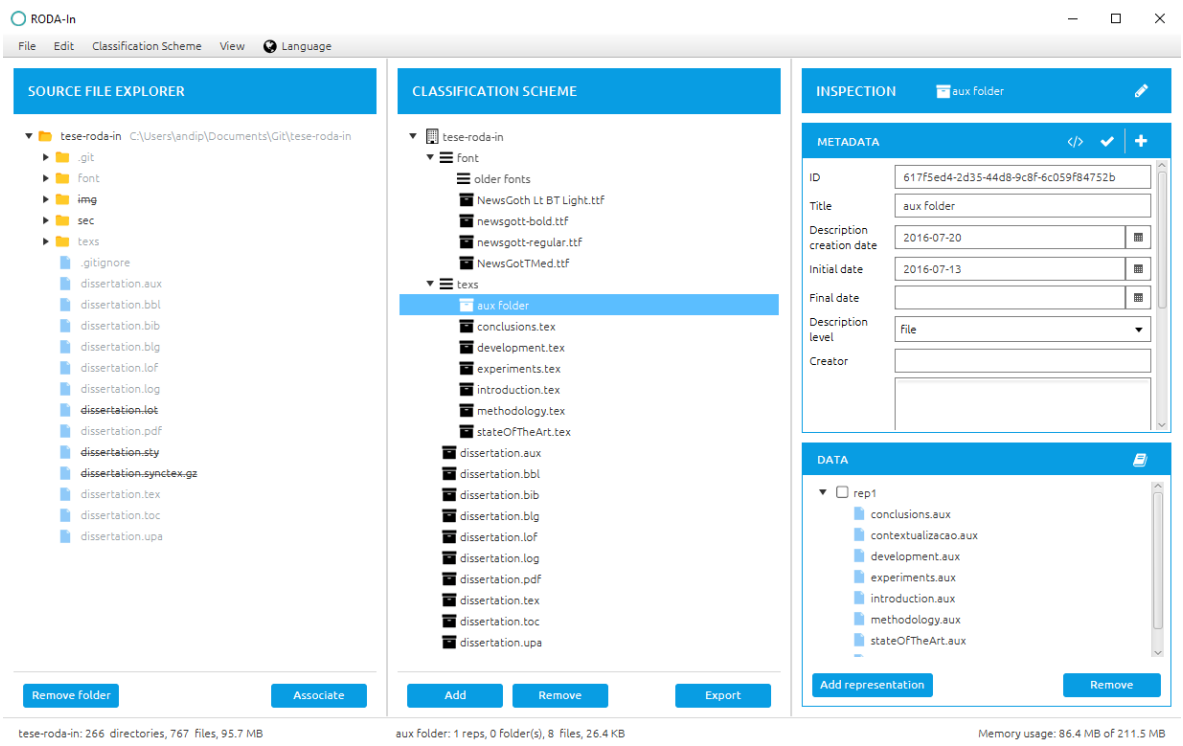


Figure 5: RODA-in with files mapped and ignored. One of the SIPs is being inspected.

There are additional steps that the user can take to tweak the output, such as preventing specific files to be added to the SIPs (e.g. Thumbs.db, DS_Store, invisible files, etc.) and editing or adding new descriptive metadata. One of the goals of this tool is to provide a simple approach to enable users to create thousands of SIPs with just a few mouse clicks, but also to enable advanced features that satisfy the needs of more demanding users.

Figure 5 shows the user interface of RODA-in 2.0. The left-most panel lists the directories and files added as source data. The middle panel is used to create and edit the classification scheme, in addition to viewing the created SIP previews. Finally, the right-most panel displays the content of the selected item from the classification scheme, enabling users to inspect and edit the created SIPs.

4.2 SOURCE FILE EXPLORER

The first step of the workflow is to choose which computer files and folders to archive and add them to the application's file explorer. This action will display the folder and its content in a folder tree structure. The folders and files are clearly distinguishable from each other and folders can be expanded in order to inspect its children elements.

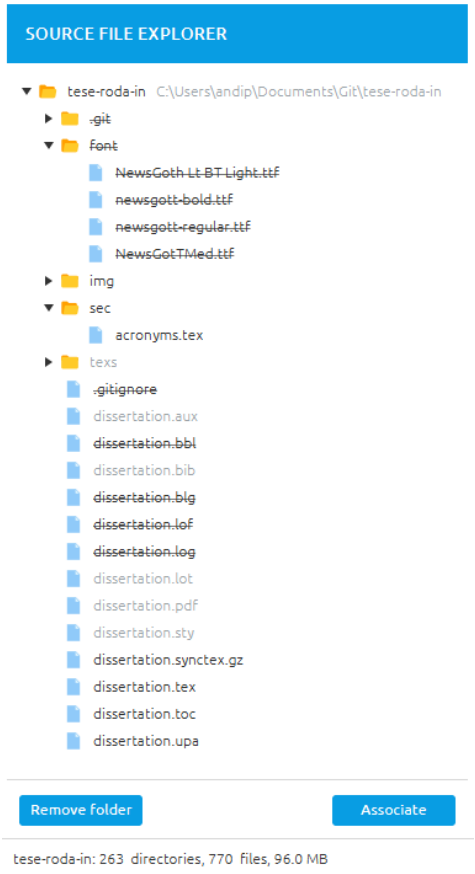


Figure 6: The file explorer with mapped, ignored and normal files.

Special attention has been dedicated to cope with large amounts of files and folders. User graphical interfaces struggle to display folder contents with millions of items. To overcome this problem, only the first one hundred items are loaded at once and a button has been included to load the next hundred items. Even if not all the items are visible at a given time, these will processed adequately if their hierarchical ascendants are selected for processing.

An item can be in one of three states: normal, ignored or mapped. **Normal** is the default state of the items and the only state from which SIPs can be created.

Items in the **ignored** state will not be used to create packages. They can be ignored based on rules defined in the configuration of the application, or manually using the "Ignore" button. These items are hidden by default, but can be made visible by turning on a menu option. Ignoring items is useful when there's a mixed source set of files and some of them are not supposed to be packaged as data, possibly because they're metadata files or simple operating system files (e.g. Thumbs.db, DS_Store, etc.).

Mapped items are the ones that have already been added to an SIP. The goal is to implement a strategy of "map and forget" in which users drag and drop items from their file

system to create SIPs after which they disappear from the user interface. This enables users to fully focus on the remaining items and thus be more productive in their SIP creation project.

As shown in Figure 6, normal items have a simple black value, while ignored items are characterized by a strike-through text value and mapped items by a grey text value. In the figure, the mapped and ignored items are visible because the user toggled them in the menu, otherwise the only visible items would be items with the normal state. The top item, named "roda-in", has the full path of the folder to the right of the name so the user avoids confusion between top level homonymous folders. Adding the full path also has the added feature of highlighting the top folders. Below the file tree, two buttons add important functionalities to the tree: "Remove Folder"/"Ignore" and "Associate". The first has two different functionalities merged in a single button: if a top folder is selected it will remove it and all its children, but if an item other than a top folder is selected, the button text will change to "Ignore" and it will mark the selected items as ignored. The second button starts the association process with the creation of a new rule, effectively working analogously to the drag and drop system. At the bottom of the panel, a footer displays the count of directories, files and size of the selected items.

4.3 Definition of the classification scheme

This optional step allows the user to define how the SIPs will be organised in a classification hierarchy. This classification can be imposed by the archive, created voluntarily by the producer, or a mix of the two. In the later case, the hierarchy is imposed by the Archive up to a given point and then further defined by the producer. There is also the option of not using a classification scheme. In this scenario, SIPs are created without holding any information about where they fit in the classification hierarchy. It is up to the Archive to decide how the SIPs should be classified after the ingest process.

The classification scheme can either be imported from file or created directly in the application. In the first option, the user must import a JSON formatted file which was either exported by RODA or RODA-in, since these are the only supported formats at the time. To create a classification plan, the first step is to add description items using the "Add" button present at the bottom of the central panel. When the button is pressed, a new description item is created and appended under the selected description item or under the root if no description item is selected or the selected item is a SIP. The levels are created by default with a metadata file obtained using the EAD 2002 template. Since the metadata file was created using a template, a form is available where the user can edit the title and description level of the item. In addition to adding items, there are two other actions available, removal and organization.

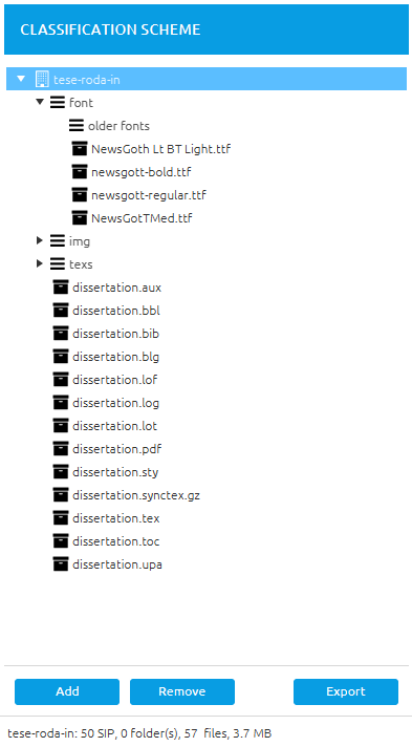


Figure 7: The classification scheme with one description level selected.

The removal process starts by selecting the items we wish to remove and clicking the "Remove" button, available at the bottom of the panel. Any selected SIPs are removed and its contents return to the normal state, so that they can be added to other SIPs. When a description item is removed, all its descendant SIPs are also removed.

By dragging and dropping selected items on top of other items, the classification scheme can be completely re-organized. The behaviour of the drag and drop system is described as follows:

- When dropping items on a description item, the selected items are added as direct children of the item.
- When dropping items on an empty section of the tree, the selected items will be added to the root of the tree.
- When dropping items on a SIP, the items are added to the children of the SIP's parent, i.e., the dropped items are at the same level of the SIP. This happens because SIPs cannot have descendants in RODA-in.
- When dropping a parent on one of its descendants the action is cancelled, otherwise there would be no connection left to the rest of the tree and the entire sub-tree (starting on the dragged item) would be lost.

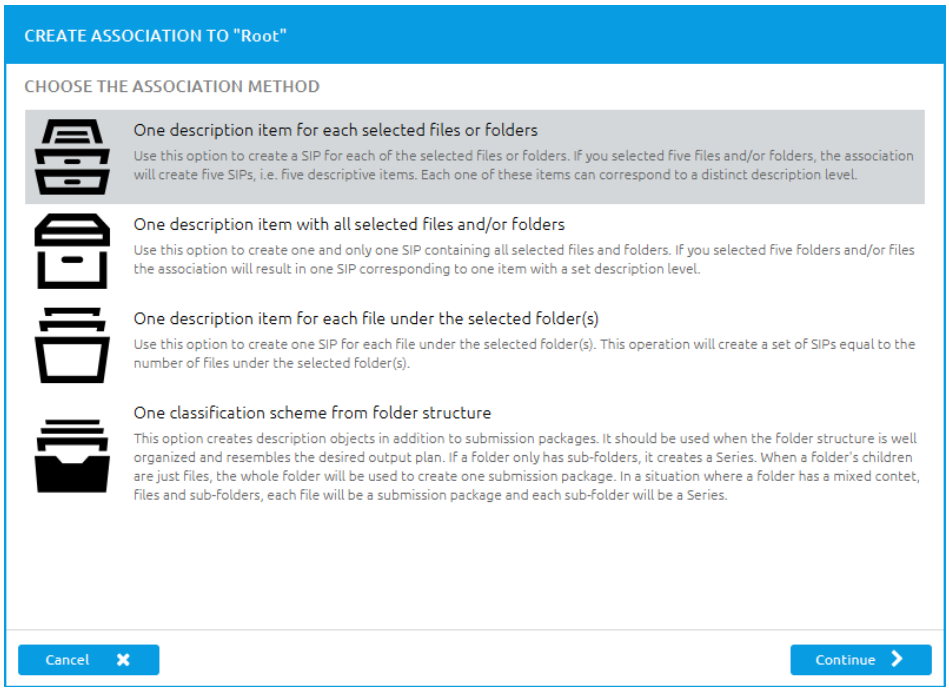


Figure 8: The panel with the four association options in RODA-in.

As can be seen in figure 7, after the classification scheme is completed, the user may opt to save it for later use using the "Export" menu option which can be found under the "Classification Scheme" menu. The rightmost button at the bottom of the classification scheme, named "Export", starts the SIP export process (which will be described in another section). Lastly, the footer displays information about the selected item: the number of SIPs, folders, files and size.

4.4 CREATION OF SIPS

The file explorer and the classification scheme are fundamental to the SIP creation workflow implemented by RODA-in. The first is used to select the data to be included in the SIPs, and the second is used to define the collections or series where the data should be placed once it's accepted by the repository.

To create SIPs, the user has to choose an aggregation rule. Start by selecting the files and folders that hold the content to be included in the SIPs and drag n' drop them on a node of the classification hierarchy. This node can either be a classification scheme node, a SIP or the root level (i.e. no specific node). After this operation, the user is presented with a set of options that will shape the way the SIPs are created.

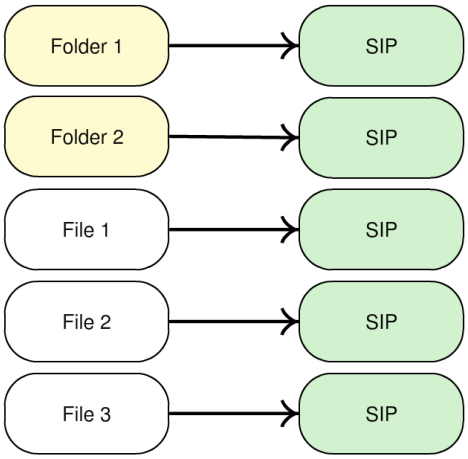


Figure 9: Folder and files mapping to SIPs using Option 1.

Aggregation rules

There are currently four content aggregation options that should satisfy the needs of most producers. More options will be added as the user community discovers other prominent patterns. The existing options can be seen in figure 8:

Option 1: One SIP for each of the selected files or folders

This option creates one SIP for each selected file or folder. It is mostly useful when producers organise records on a per folder-basis, i.e. each folder contains all the files that are relevant to a given record.

For example, if the input is composed of 3 files and 2 folders, this association option will create 5 SIPs, one for each of the 3 files and one for each of the 2 folders. This is a N to N mapping between the input data and the output SIPs. See figure 9.

Option 2: One SIP containing all selected files and/or folders

The second option creates one and only one SIP containing all the files and folders from the input. It's useful when the producer wants to create one single SIP containing a large number of files and folders (potentially the entire year worth of records or a hard drive).

For example, if the input is composed of 5 files and 2 folders, this association option will create 1 SIP containing the entire set of selected items. See figure 10.

Option 3: One SIP for each file under the selected folder(s)

This option creates one SIP for each file included in any of the selected items. The files are found recursively inside the folders, creating a set of SIPs equal to the number of files found. It's useful when each individual file represents an entire record.

For example, if the input is composed of 3 files and 2 folders with 10 files each, this association option will create 23 SIPs, one per each selected file and per each of the 10 files under each of the 2 folders. See figure 11.

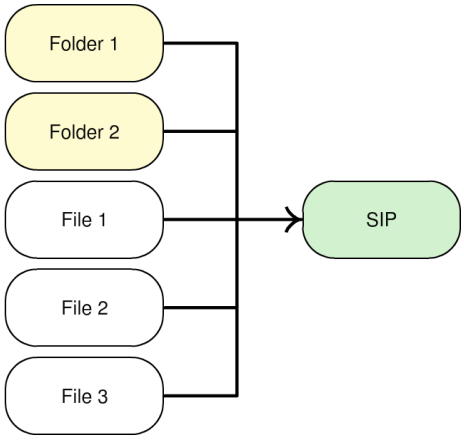


Figure 10: Folder and files mapping to SIPs using Option 2.

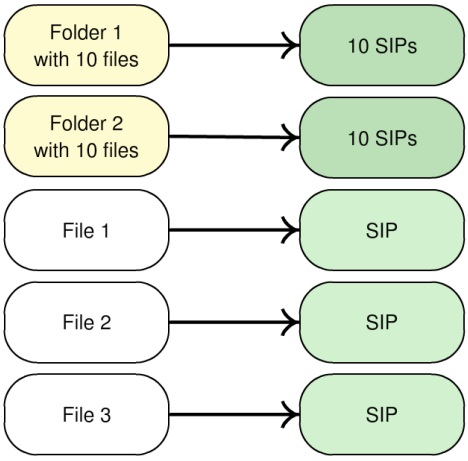


Figure 11: Folder and files mapping to SIPs using Option 3.

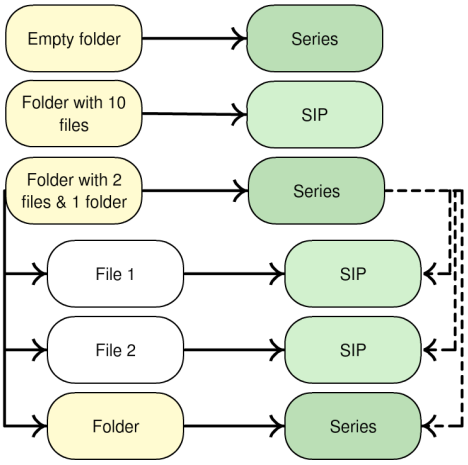


Figure 12: Folder and files mapping to SIPs using Option 4.

Option 4: Create classification scheme and SIPs based on folder structure

This option creates a classification scheme and SIPs containing data based on the original folder structure that holds the data.

This option is useful when the producer maintains a well organised folder structure that resembles the desired output classification scheme. Leaf folders that contain files are treated as SIPs while ascending folders without files are used to create nodes in the classification scheme. See figure 12.

The rules in place are as follows:

- If a folder has sub-folders and no files, it creates a node on the classification scheme.
- If a folder only contains files and no folders, the whole folder will become a SIP.
- If the folder has mixed content, i.e. files and sub-folders, each file will be a SIP and each sub-folder will be a node in the classification scheme.

Patterns for metadata association

SIPs may contain data and metadata. The metadata is included in each SIP by means of association rules. These association rules define how descriptive metadata selected from the file system and included inside each SIPs.

There are currently four available metadata association options, but the implementation enables more to be added in the future. The available options are shown in figure 13:

Option 1: Metadata based on a template

In some cases, producers may not have metadata and they expect the SIP creation tool to facilitate this activity. RODA-in comes with a metadata templating system that enables

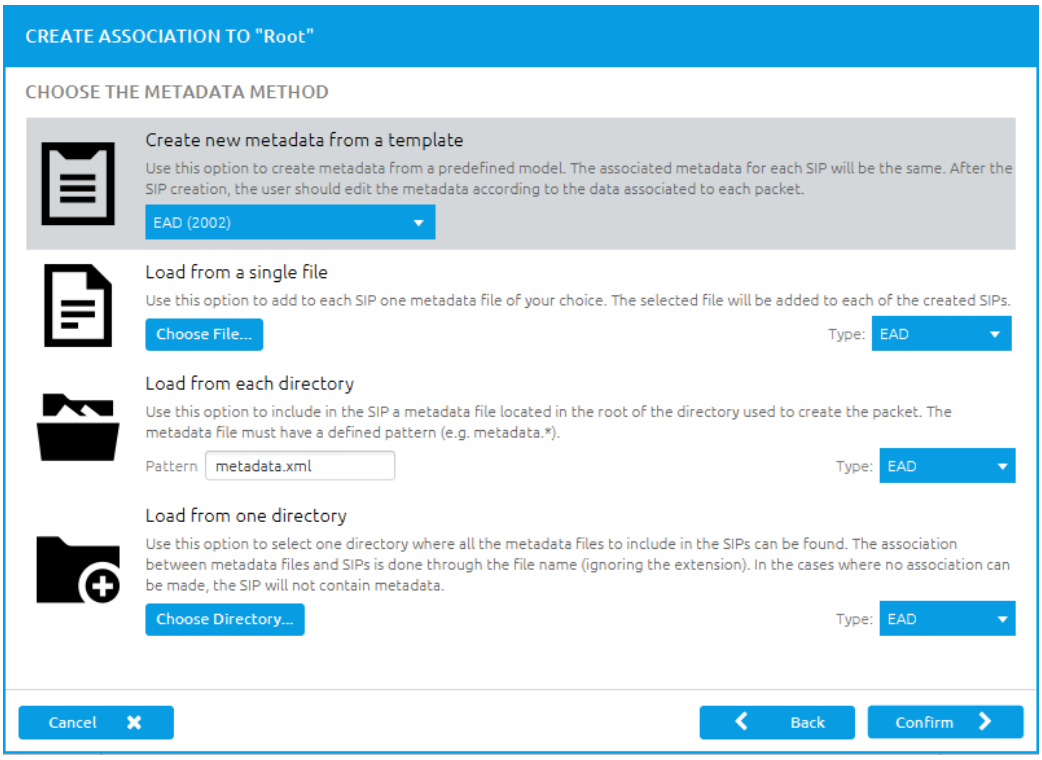


Figure 13: The panel with the four metadata options in RODA-in.

users to add their own descriptive metadata schemas to be rendered as easy-to-use input forms.

The templating system implemented by RODA-in enables the user to define the base structure of the descriptive metadata to be included in SIPs, define fields to be displayed and edited in the user interface and fields that should be filled with a predefined value (e.g. institution name) or filled by a auto-generated value (e.g. current date).

By default, RODA-in comes with templates for EAD 2002, EAD 3 and Dublin Core that users are free to change and adapt to their own needs.

Some fields from the template are pre-filled by the application:

TITLE The title is inferred using the description item's content, like the name of a folder or file in a SIP.

CURRENT DATE Sets the value as the current date in the format yyyy-MM-dd

ID Generates a random UUID

LEVEL Sets the value as the item's description level.

PARENTID The UUID of the item's parent.

LANGUAGE The language of the system the application is running on.

Option 2: Same metadata in every SIP

This option should be used when the user wants to apply the same metadata file to every package created. The input must be the metadata file that will be added to each of the SIPs to be created.

Option 3: Load metadata from data folder

This option takes a user-defined pattern [1] to match file names under the data folder and add the identified files to the SIP as descriptive metadata. This option can add more than one metadata file to the same SIP, if the pattern matches multiple files. When no files match the pattern, the SIP will be created without metadata.

Option 4: Load metadata from a folder

When all the metadata files are located in a single folder, the producer can use this option to selected and include one metadata file in each of created SIPs. The tool will try to match the name of metadata file with the name of the SIP (based on the data folder name). For a match to occur, the name of the metadata file (without the extension) and the SIP must be the same. Just like the metadata option number 3, if no matches are found, no descriptive information will be added to the SIP.

4.5 HANDLING EXCEPTIONS TO THE RULE AND SIP ENRICHMENT ACTIVITIES

After creating SIPs, users can opt to manually enrich the SIPs or handle exceptional cases manually. This is an optional step, nevertheless it contains some important features that should not be overlooked.

When SIPs are created using the metadata template option, users are able to edit metadata via an input form. Additionally, the XML source of the metadata file can also be displayed and edited on the user interface.

In addition to editing metadata files, it's possible to add and remove metadata files. There are three options available when adding a new metadata file: 1) new metadata based on an existing template, 2) new metadata based on an existing file, and 3) an empty metadata file. To avoid conflicts, two metadata files of the same description item cannot have the same name, so new files are rejected if this happens. As a consequence of this, two files based on the same template cannot be present in the same description item.

Besides changing the descriptive metadata in the SIP, the producer can also edit the data content of the SIP. The content is divided in "representations" and the user is able to add and remove these as long as there is at least one representation left in each SIP. Removing files and folders from the SIP content is also possible. These files will return to the "normal"

1 A glob: http://man7.org/linux/man-pages/man7/glob.7.html

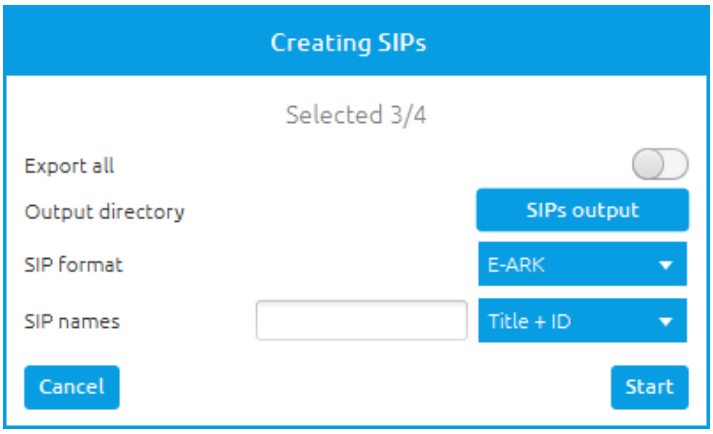


Figure 14: The pre-export panel.

state and will become available in the file explorer to be included in another SIP. A drag n' drop mechanism of the data folder structure enables rearranging the content of the SIP and move files between representations.

The RODA-in tool also enables adding files to the "documentation" folder as defined in the E-ARK SIP format. This acts as a transfer place holder of Representation Information (as defined by OAIS). (CCSDS, 2012)

4.6 EXPORTING SIPS

The last step in the SIP creation process is to export the mapped SIPs to an output format to be submitted to the repository for ingest. It's important to note that RODA-in does not check if the SIPs are valid before exporting them and this process should be performed on the repository or with an external tool.

The producers can personalize the export process to be tailored to their needs. The first decision is whether to export all the SIPs present in the classification plan or only the selected ones. By default, the only exported SIPs are the selected items when the user clicks the "Export" button, but that can be overriden using the "Export all" toggle button. Another required field is the destination of the SIPs once they're exported, which is achieved using a simple OS folder chooser, familiar to most users. It's also at this stage that the export format/type ust be chosen, at the moment, RODA-in supports two: E-ARK SIP and BagIt.

In order to personalize the exported SIP names, RODA-in offers 3 types:

- **ID** uses the unique identifier of the SIP as its name. Example: ID41b3e38d-6786-4abf-afe6-359993ed58c8.zip

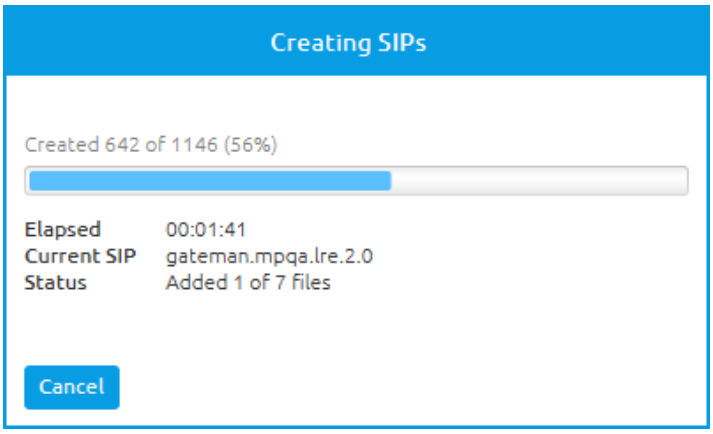


Figure 15: The export process panel.

- **Title + ID** appends the SIP's title (which can be defined using descriptive metadata) to the unique identifier. Example: Day3PMSessionPT - ID1a7104b2-53f5-43e0-9447-32c4de5f16d9.zip
- **Title + Date** appends the SIP's title to the export timestamp. Example: Day3PMSessionPT - 2016.07.05 22.27.56.793

In addition to these 3 types of name construction types, the user may also prefix the name with a String to provide additional context, for example, prefix the names with "Transcripts" could output "Transcripts - Day3PMSessionPT - ID1a7104b2-53f5-43e0-9447-32c4de5f16d9.zip".

The SIP names being well-formed and personalizable is crucial. It offers the producer and the Archivist ways of ordering the SIPs and may provide some context before opening them. Additionally, it eases communication in case SIPs are rejected and need to be fixed by the producer.

Once all the configurations are complete and the export process is started, the application begins notifying the user of the process progress. In addition to providing the elapsed time and the number and percentage of SIPs already exported, it also shows the progress of the current SIP. This is specially useful for large SIPs, which may take a long time to export and the progress bar doesn't advance. The user may also cancel the export process, however, already exported SIPs will not be deleted.

```
<base directory>/
├── data/
│   └── [payload files]
├── [optional tag directories]/
│   └── [optional tag files]
├── bagit.txt
├── manifest-<algorithm>.txt
└── [optional additional tag files]
```

Figure 16: Basic bag structure.

The BagIt format

BagIt is "a hierarchical file packaging format for storage and transfer of arbitrary digital content. A 'bag' has just enough structure to enclose descriptive 'tags' and a 'payload' but does not require knowledge of the payload's internal semantics." (Boyko et al., 2011).

The minimum structure of a valid bag is described in Figure 16. The base directory may have any name.

The "bagit.txt" file declares the BagIt version and the encoding of the tag files. The "manifest-<algorithm>.txt" file is a payload manifest that lists all the payload files and their checksum using an algorithm like MD5 (Rivest, 1992) or SHA-1 (Eastlake and Jones, 2001). The data folder is where the payload files are saved.

One of the more important optional tag files is bag-info.txt because that is where all the bag's metadata will be. Since there is no concrete location in a bag to send metadata in, RODA-in deposits all the metadata files from the SIPs internal representation to this file in a key-value format, where the key is "metadata.<file_name>" and the value is the file's content. It is up to the repository to parse this information and divide it in files again.

There is some information supported by RODA-in that is not directly mapped when exporting to the BagIt format since there is no adequate placeholder for "representations" or "documentation". Nevertheless, this format can cope with the simpler cases where this complexity is not needed and would not even be supported on the target digital repository.

The E-ARK SIP format

The E-ARK SIP was developed in the E-ARK project because "there is currently no central SIP format which would cover all national and business needs as identified in the E-ARK Report on Available Best Practices". (Kärberg et al., 2014)

A simple description of the E-ARK IP structure is available in Figure 17.

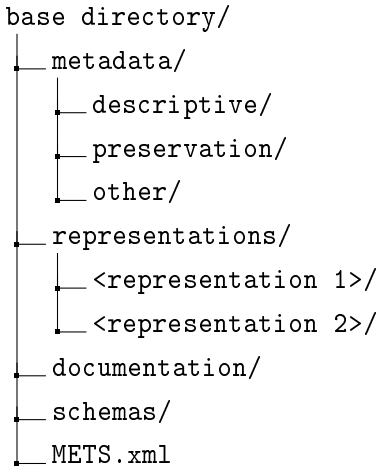


Figure 17: Basic E-ARK IP structure.

The E-ARK SIP format is more complex than BagIt. It has distinct folders for metadata, data (with support for "representations"), documentation and schemas. The METS.xml is a "mandatory metadata file which includes core information needed to identify and describe the structure of the package itself and the rest of its components." (Kärberg et al., 2016) This file is based on the METS standard and is presented as a profile with 5 main sections:

- "<metsHdr> - METS header (metadata about the creator, contact persons, etc. of the IP)."
- "<dmdSec> - descriptive metadata (references to EAD, EAC-CPF, etc.)."
- "<amdSec> -administrative metadata (how files were created and stored, intellectual property rights, etc.)."
- "<fileSec> - file section, lists all files containing content (may also contain metadata about files)."
- "<structMap> - structural map, describes the hierarchical structure of the digital object and the whole IP (i.e. object + metadata)."

Currently, all the metadata is added to the E-ARK SIP as descriptive metadata. The documentation is added exactly as displayed in the application, keeping the full folder structure and all the files. The schemas folder is by default populated with the METS schema file which depends on the XLink schema, which is also added. Moreover, the schemas from the metadata templates used are also added to this folder, given that they are set in the template's configurations.

Each representation also has its own folder structure that is similar to the overall SIP structure, as shown in Figure 18.

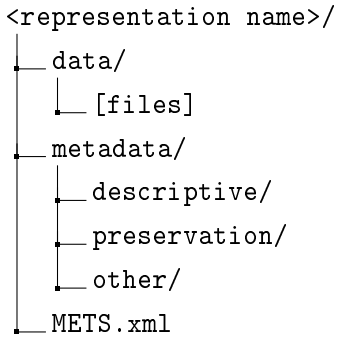


Figure 18: Representation structure.

At the representation level, the "METS.xml" file is optional and should be used to handle scalability issues since it describes the structure and content of the representation only. The data folder is where the files that will be archived are stored. If all the metadata is stored at the IP level then there's no need to use the Metadata folder at the representation level. Aside from this folder being optional and user feedback showing that it isn't an essential feature, RODA-in doesn't yet support adding metadata or documentation for specific representations, only at the IP level.

4.7 OTHER FEATURES

Besides the described features, RODA-in has other components to help the producer create SIPs like descriptive metadata validation, templates, forms and documentation.

Internationalization

The application is currently supported in four languages: English, Portuguese, Spanish and Hungarian. The Portuguese and English content was written by the RODA-in team, while the Hungarian content was translated by Szatucsek Zoltán, from the Hungarian National Archives.

The language will be determined by the locale of the operating system that runs the application. However, this setting can be altered by a property on the configuration file or using a menu from within the application. Each of these actions requires a restart for the changes to be applied.

Should a required message be unavailable in a language, the application tries to use the corresponding English message.

Templating system

The templating system is based on the widely used semantic templates library Handlebars.[2] A template is a string that contains any number of tags. Tags are indicated by the double curly braces that surround them. {{person}} is a tag, as is {{#person}}.[3] Each of these tags corresponds to a text field on the form, where the user can set the value that will replace the tag when the metadata file is exported. There are some tags that the application recognizes and automatically sets them with auto-generated values.

The templates are completely customizable since it's possible to not only edit their content but also add new templates or remove those that are unwanted. All of the user customizable data, like the configurations file, the templates and the logs are in the application folder that can be found in the user's home directory in Unix systems or in the "My documents" directory on Windows. To add a template the user needs to follow some easy steps:

- Add the file with the template to the "templates" folder
- (optional) Add the schema that validates the template to the "schemas" folder
- Edit the configuration file

In the configuration file, the archivist needs to insert data to tell the application which file, title and type should be used in the new template. Another required field is the unique identifier, which in the example below will be "ead".

```
metadata.template.ead.file=ead2002.xml
metadata.template.ead.title=EAD
metadata.template.ead.type=ead
```

After this configuration block has been added, all the user needs to do is add the template to the active templates list and restart the application:

```
metadata.templates = ead,ead3,dc,eadDGLAB
```

Since it's tedious to edit these files by hand once they are added to SIPs, a way to create forms based on the fields of the templates was added. Using the powerful Handlebars engine, anyone can create template files with the necessary information to create the form. In addition to a simple tag, (e.g. person), we can now add options which will modify the way each field is created. These options are key-value elements, e.g. `title="SIP creation using RODA-in"`, where the key is the name of the option and the

2 http://handlebarsjs.com/
3 https://github.com/janl/mustache.js

value is the value that will be given to that option. The tags should be declared at the top of the file to keep it organized, although this is not mandatory.Furthermore, the options are not required, the form is still created with a simple tag, which creates a standard text field with the field's name/identifier as its label.

The available options that alter the fields created for each tag are:

VALUE the predefined value of the field

ORDER the order of the field

TYPE the type of the field. The possible values are:

- **text** - text field
- **text-area** - Text area. Larger than a field text.
- **date** - text field with a date picker
- **list** - list with the possible values (combo box)

LIST List with the possible values that a field can have. Usable when `type="list"`. The format is a JSON array. Example: `[option A, option B, "option C"]`

LABEL The label that appears to the left of the field.

MANDATORY If set to true the label is styled in bold to draw attention.

HIDDEN If set to true the field is hidden

AUTO-GENERATE Fills the value with one of the available generators. Overrides the value option:

- **now** - the current date in the format year/month/day
- **id** - generates an identifier
- **title** - generates a title
- **level** - adds the current description level
- **parentid** - adds the parent's id, if it exists
- **language** - adds the system language, based on the locale. Example: "português" or "English"

Listing **??** shows a template of Simple Dublin Core:

```
<?xml version="1.0"?>

<simpledc xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../schemas/dc.xsd">
```

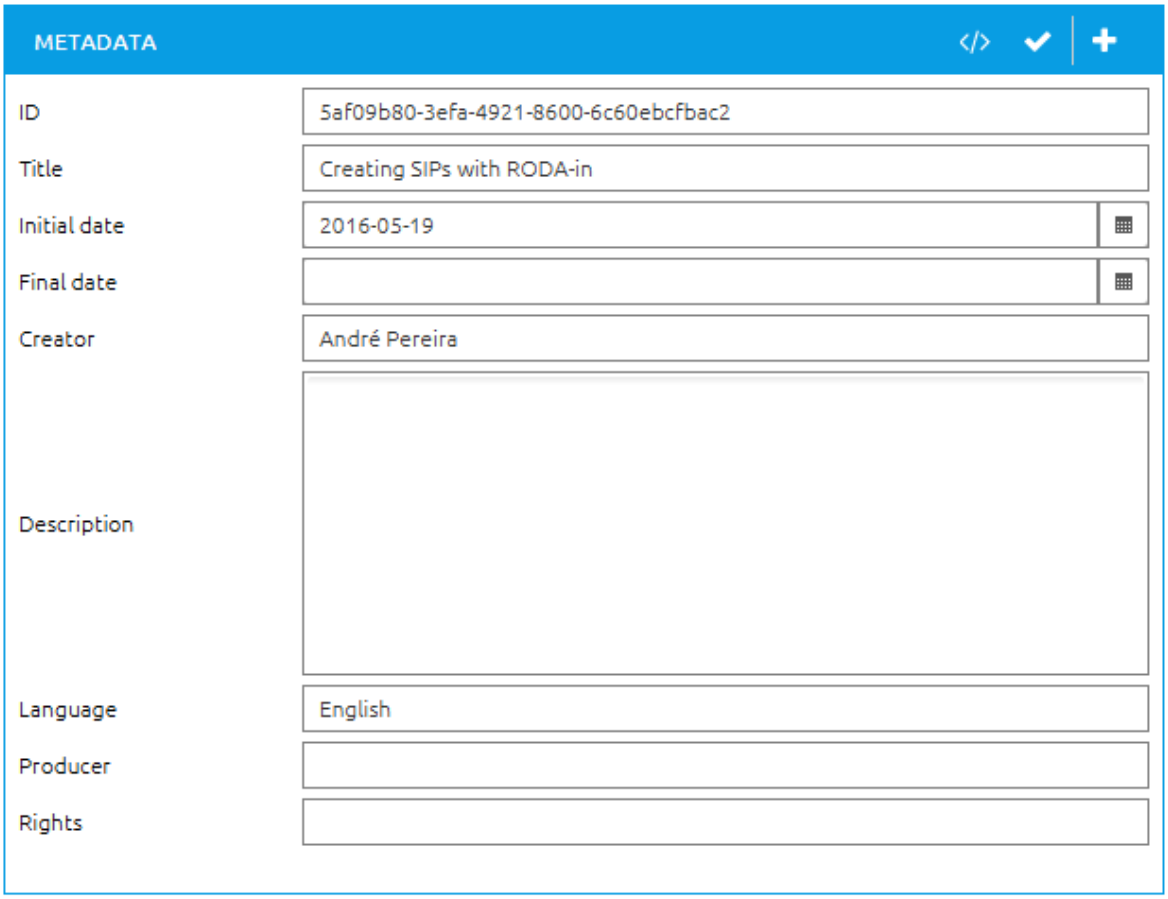


Figure 19: The resulting form of the Simple Dublin Core template

```
<title>{{title order=2 auto-generate='title'}}</title>
<identifier>{{id order=1 auto-generate='id'}}</identifier>
<creator>{{creator}}</creator>
<date>{{dateInitial order=3 type='date'}}</date>
<date>{{dateFinal order=4 type='date'}}</date>
<description>{{description type='text-area'}}</description>
<publisher>{{producer}}</publisher>
<rights>{{rights}}</rights>
<language>{{language auto-generate='language'}}</language>

</simpledc>
```

Listing 4.1: A template of Simple Dublin Core

Figure 19 shows the resulting form based on the above template. It's clear that the ID field is the first in the form, followed by the title field, as specified in the *order* fields of the

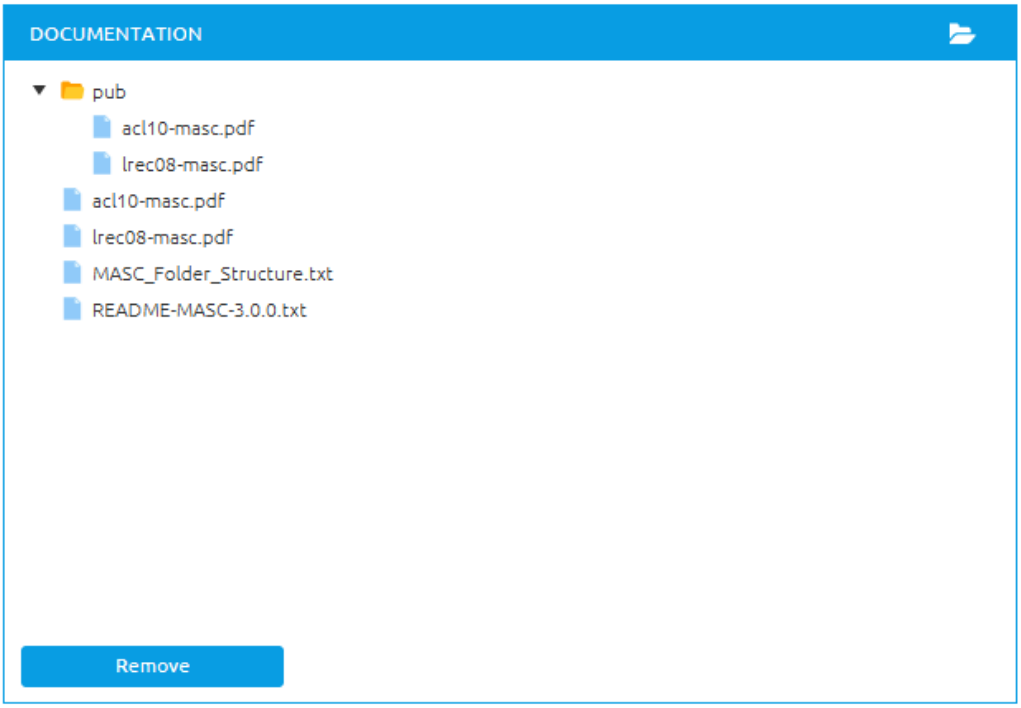


Figure 20: The documentation section of a SIP.

template. The *Initial date* and *Final date* fields have a date picker to facilitate choosing the desired date, but it's also possible to write it in the same text field. The *Description* field is larger than the others since it's type is "text-area". The *ID*, *Title* and *Language* fields were prefilled because the auto-generate option is defined in each of them.

Even though the template doesn't specify a label for the fields, the application has a small database of possible fields and their translations. Using the identifier of the tag, e.g. dateInitial, the application checks if it has a language entry for this identifier in it's language files and sets the label with that value. If there's no translation, the identifier is used in its place. All the fields in the default templates of RODA-in have an entry in the language files.

SIP documentation

Documentation can be added to SIPs to complement the information that is being archived in the Data section. Files and folders can be added from the application's file explorer or from the operation system's file explorer using a drag and drop system. Unlike when files are added to the Data section of SIPs, adding documentation does not set the state of the files as mapped, so an archivist can add the same file to the documentation of any number

of SIPs. After being added, the files can be removed and rearranged in the same way as in the Data section.

Metadata types

The metadata types are essential when adding metadata to a description level, since it's the property that describes each metadata file. This property is set in the METS file of the E-ARK SIP and helps the repository that ingests the SIP know the type of the metadata and if it should handle it in a special way. In the user interface, these types are displayed when selecting the metadata association option for all but the metadata from template option, because we've already set it in the configuration.

Much in the same way as the metadata templates, metadata types are declared in the application's configuration file. The active types are disclosed in a list format with the types' identifiers. The three types bundled with the application are: EAD, Dublin Core and Key-Value.

```
metadata.types = ead2002,ead3,dc,keyvalue
```

The description of each type is accomplished using a type identifier, the value and version that will be set in the METS file, the title for display purposes and an optional schema for validation.

```
metadata.type.ead2002.title=EAD 2002
metadata.type.ead2002.value=EAD
metadata.type.ead2002.version=2002
metadata.type.ead2002.schema=ead2002.xsd
```

Metadata validation

Provided that the schema is added to the type definition, RODA-in provides another useful feature: metadata validation. This feature is only available to metadata files created from templates and the template's type must have a schema correctly added in the configuration. In the case where a user is editing a form, with the push of a button the tags from the template are temporarily replaced with the corresponding values and the resulting file is validated against the schema. If the file cannot be validated successfully, the application displays the detailed error message and the location of the error, otherwise it only shows a success message.

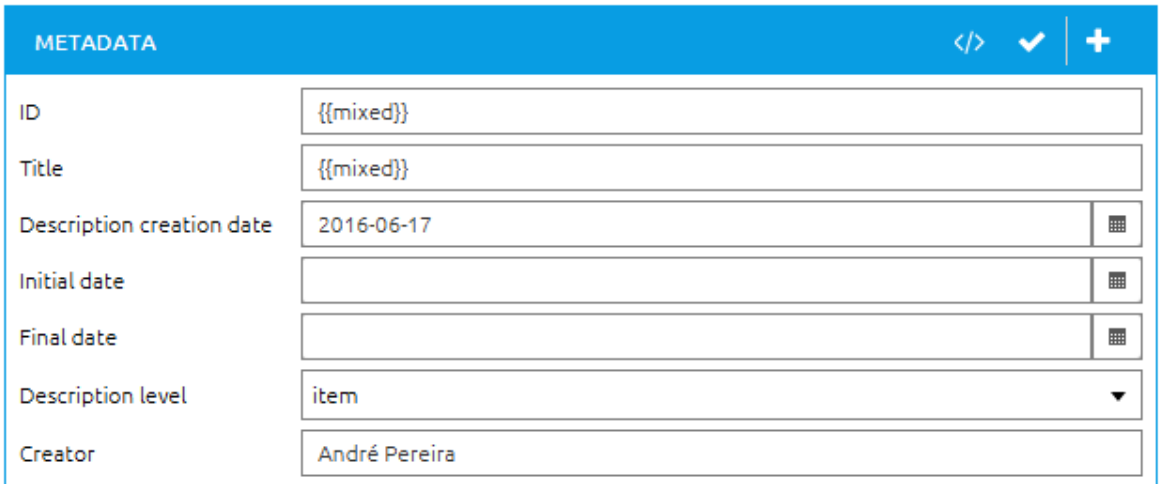


Figure 21: The view of the inspection pane when multiple items are selected.

Assignment of metadata to multiple items

Editing metadata of multiple items was a highly requested feature, since the users regularly need to apply a value to the same field or group of fields. By selecting multiple items of the classification plan tree, be it SIPs, description levels or a mixture of both, the inspection panel opens the multiple item edition view. This view, enables the user to add additional metadata files to all the selected items.

The metadata files added to the list will be added to all the selected items from the classification plan. If there are any conflicts, i.e. an existing metadata file has the same name as a new one, the existing file will be replaced. The only exception is when the files are the same template. In this case, the values of the new template will override the old ones. The only cases where the old value is kept is if the value of a field is auto-generate or mixed.

By default, the {{auto-generate}} text is set in the ID, Title and Description level fields of a new template, since these fields are less likely to be edited. This tag tells the application to ignore the content of that field and use the value already available in that item.

The {{mixed}} text is part of a special case of the the edition of multiple items. Whenever all the selected items have one and only one metadata file and that file was created using a template, the view shows a special form. Since all the information is structured in the same way and the fields are the same in every item, it's possible to edit each field individually while having some context of how the other fields will be affected by the changes. If a field has the same value in all the items, it will have that value set once it's added to the form. Otherwise, the field will be set with the {{mixed}} text, which signals the application that the field must not be changed when the user saves the changes. In figure 21, the Title field

has mixed content, so it won't be modified once the changes are saved. However, if we wanted all the items to have the title "Editing multiple items", we simply need to remove the current text and input the new one, effectively overriding the values currently saved in each item. This approach has been widely used in music library management applications such as iTunes [4], MediaMonkey [5] and MP3Tag [6].

4.8 TECHNOLOGIES

This section presents most of the technologies used in the development of this project. Each technology has a small description, why it was chosen and its alternatives.

4.8.1 *Platform*

The programming language and platform used for the development of the tool had to be carefully chosen to satisfy all of the proposed objectives. The requirements for the development platform are:

1. File system – The platform must have full control over the file system
2. Performance – Fast processing and I/O is required. The file system should be accessed using streaming or any other memory friendly method of listing and accessing the files.
3. Multi-platform - The final application should be runnable in all the major operation systems (Windows, Mac OS and Linux)
4. Graphical user interface – Support for a graphical user interface is required, either by the use of plugins/extensions or native to the technology.
5. Usability – Ease of usage of the end product

Since web frameworks are in vogue and the number of web technologies is expanding every day, these were the first platforms analyzed. The backend (or server side) for the tool could be a simple set of static pages, built using one of many fast prototyping frameworks available like Ruby on Rails [7], Sinatra [8], Django [9], Laravel [10] or NodeJS [11]. The frontend (or

4 http://www.apple.com/itunes/
5 http://www.mediamonkey.com/
6 http://www.mp3tag.de/en/
7 http://rubyonrails.org/
8 http://www.sinatrarb.com/
9 https://www.djangoproject.com/
10 https://laravel.com/
11 https://nodejs.org/en/

client side) would need to have a powerful way of accessing the file system in addition to complete and useful components, so the choice would have to be one of the many JavaScript frameworks available, like AngularJS [12], Ember.JS [13] or React [14].

The web application approach is multi-platform, has great performance and has a user interface, but falls short on the file system requirements. Since JavaScript runs on a sandbox, it cannot create files outside of its environment, which is required to create the SIPs. One way to overcome this problem would be to have all the logic of the SIP creation running on the frontend (which files, metadata and documentation are in a SIP) and upload the data to the server, which would create the SIP and make it available for download. This approach has several problems: the bandwidth usage and the load on the server would be enormous; for large SIPs the upload and consequent download would possibly take an unacceptable amount of time; the possibility of data corruption is larger since we're transferring it multiple times. Since the web application approach has several problems and doesn't fulfill all the requirements, it was discarded.

Excluding the web applications, the tool would have to be developed using a technology that produces an offline application. The choices are nearly endless: Java [15]; Scala [16]; C++ [17]; C# [18]; Python [19] or Ruby [20]. Every one of these examples fulfills all the requirements set for the application, some better than others. After analyzing all the options, the choice fell to Java, since it's the easiest to run in multiple operating systems with minimal preparation and because it's a proven language in the enterprise world.

Java

Java is an object-oriented computer programming language which was originally released in 1995 by Sun Microsystems (which has been acquired by Oracle Corporation). The code is compiled to bytecode that can run on any Java virtual machine (JVM) [21], regardless of operating system. The language is currently on version 8 [22] (released in 2014), although version 9 is scheduled for release in 2017 [23]. Version 8 introduced some much needed features that brought Java to the forefront of functional programming, like Scala has been doing for some years. Lambda expressions, parallel operations and a new Stream API helped fix issues that many developers had with the language. This version was also the

12 https://angularjs.org/
13 http://emberjs.com/
14 https://facebook.github.io/react/
15 https://java.com/
16 http://www.scala-lang.org/
17 http://www.cplusplus.com/
18 https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx
19 https://www.python.org/
20 https://www.ruby-lang.org/en/
21 http://www.oracle.com/technetwork/java/intro-141325.html
22 http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html
23 http://openjdk.java.net/projects/jdk9/

first to be packaged with JavaFX, which intends to replace Swing as the GUI library for Java SE.

JavaFX

JavaFX is a platform to create rich internet applications and desktop applications, introduced in 2007 by Sun Microsystems. The developer can use an XML-based markup language, called FXML, to describe the structure that will build the user interface, separately from the application logic. In a perspective of Model View Controller (MVC), the FXML files can be considered the View component. Oracle also made available a scene builder to easily create and edit these files, although there have also been a few efforts from the community in this area, like the open source tool Gluon Scene Builder [24].

In contrast to FXML, the developer may choose to create the user interface programmatically, extending the available classes in JavaFX. Since I felt that by creating the UI programmatically there were a greater number of available options, I decided to use that technique.

To style the application, the developer can use CSS much in the same way it is used in the elements of the HTML DOM, where styles are first applied to the parent and then to its children [25]. Although CSS can be used, it still has a few limitations since the parser is not fully compliant with CSS. For example, the pseudo-classes *:first-child*, *:lang* or *:active* are not supported, likewise the *:after* and *:before* pseudo-elements cannot be used. The properties are prepended with "-fx-", so what would be a "background-color" property in normal CSS would be "-fx-background-color" in JavaFX.

Maven

Apache Maven [26] is a software project management and build tool, used primarily for Java, C#, Ruby and Scala. It was introduced in 2004 and is currently on version 3. The project object model (POM) is an XML file which describes the software project being built, its dependencies, build order and required plug-ins. The dependencies are dynamically downloaded from repositories such as the Maven 2 Central Repository or from user created repositories. Alternative technologies like Gradle [27], sbt [28] or Apache Ant [29] could also have been used, but I felt Maven was the most adequate choice. It also has the advantage of allowing any developer to download the code and build the application in few easy steps.

24 http://gluonhq.com/open-source/scene-builder/
25 https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html
26 https://maven.apache.org/
27 http://gradle.org/
28 http://www.scala-sbt.org/
29 http://ant.apache.org/

Using plugins, I was able to add automated testing to the application, as well as automatic packaging to JAR and Windows executable (EXE).

4.8.2 *Version control*

A version control system (or revision control system) is a repository of files where every change made to the source is tracked, along with who was responsible for the change and a description of the change. These systems are essential for distributed and collaborative development and are useful for everyone, from very large teams to solo developers. There are three different models of version control systems:

- Local Data Model - the simplest form of version control where developers have access to the same file system.
- Client-Server Model - In this model, the developers use a single share repository of files available on a network (local or the Internet).
- Distributed Model - Each developer works directly on its own local repository and the changes are shared between repositories at a later step. This was the model used on this project, since it's the one used by Git.

Git

Git is an open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency [30]. Every Git working directory (the directory that contains all the code and resources) is a repository with complete history and full version-tracking capabilities. The user uses commits to save the progress to the local repository, which can later be synchronized with a remote repository. The basic workflow of git uses branches and merges which enables a team of developers to work on the same project at the same time, while keeping the version history for backtracking, if needed. Alternative version control systems are Apache Subversion and Mercurial, among others.

GitHub

GitHub [31] is a website which provides a web-based graphical interface to explore Git repositories and offers all of the distributed revision control and source code management functionality. Furthermore, it also provides access control and multiple collaboration features such as bug tracking, wikis and task management. This tool was frequently used during the development of this project and the public repository can be found in the footnote. [32]

30 https://git-scm.com/
31 https://github.com/
32 https://github.com/keeps/roda-in

Alternative tools are BitBucket [33] and GitLab [34]. To extend the functionality of the task management system I decided to use ZenHub [35], which adds task boards to implement an Agile pipeline. This pipeline consisted of Backlog, To Do, In Progress, Done and Closed.

4.8.3 *Continuous integration*

Travis CI

Travis CI [36] is a distributed continuous integration service used to build and test software projects hosted at GitHub, free for open source projects. To configure the service, a file named *.travis.yml* must be added to the root directory of the repository. In this file the developer can specify parameters such as the programming language used, the desired building and testing environment, dependencies of the environment, among others. Some of the languages supported are C++, C#, Ruby and Java. Once new commits are pushed or a pull request is submitted, Travis CI will build the project and run the automated tests. Once the process is complete, the GitHub page is updated with the result and the administrators are notified.

In this project, Travis was only used to build the application. The tests were skipped because the support for JavaFX testing in Travis is not the best. Although Travis CI can have an X server, [37] the results weren't deterministic since some tests would fail randomly.

Jenkins

Jenkins [38] is an open source continuous integration tool, with support for most version control tools like Git. The basic platform can be extended by plugins that add integration with many other services like Maven projects, languages other than Java, testing frameworks and code analysis tools. Since there's no distributed and free Jenkins service, unlike Travis CI, I used the instance that KEEP Solutions offers to its collaborators. Since the configuration of the Jenkins instance was available to me, I was able to add the needed dependencies to enable the tests to run in an adequate environment.

33 https://bitbucket.org/
34 https://about.gitlab.com/
35 https://www.zenhub.io/
36 https://travis-ci.org/
37 https://www.x.org/wiki/
38 https://jenkins.io/

4.8.4 *Code quality*

SonarQube [39] is an open source platform for continuous inspection of code quality. It provides fully automated analyses and integrates with Maven projects and Jenkins. In addition to being expandable with the use of plugins, it also covers the 7 axes of code quality: Architecture & Design, Duplications, Unit tests, Complexity, Potential bugs, Coding rules and Comments. One of its main metrics is the SQALE method (Letouzey, 2012) which enables the evaluation of an application's technical debt in the most objective, accurate and automated way possible. This method rates the quality of an application from configurable source code requirements and provides an objective view of the application quality. Currently, RODA-in has a SQALE rating of **A**, the best value for a project.

4.8.5 *Other*

Transifex

Transifex [40] is SaaS translation platform which targets technical projects with constantly updated content, such as software projects. Some of its main features are: ability to download content, translate it offline and then upload it; Translation Memory that reduces the effort when having to translate something already translated somewhere else; messaging and notification system for keeping a translation team informed. It also offers integration with GitHub, but it wasn't used in this project.

Testing

The testing in RODA-in was possible due to JUnit[41] and TestFX[42]. JUnit is part of the xUnit architecture for unit testing frameworks which targets the Java programming language. The basic components of the framework are: test cases, test runners, test suites, test executions and assertions. TestFX is library for simple and clean testing of JavaFX applications that features a fluent and clean API, flexible setup and cleanup of JavaFX test fixtures, robots to simulate user interactions and a collection of matchers to verify expected states. Actions like drag and drop an item, click a button, selected a node or write text are essential to test a primarily user interface oriented application, such as RODA-in.

39 http://www.sonarqube.org/
40 https://www.transifex.com/
41 http://junit.org/
42 https://github.com/TestFX/TestFX

4.9 DECISIONS AND IMPLEMENTATION

This section will describe the decisions made during development and detail the implementation of the major functionalities of RODA-in.

4.9.1 *Tree views*

One of the most common elements in RODA-in are tree views, which are a graphical control element that present information in an hierarchical view. Tree views are a great way to navigate file system directories and are well known by most users since they're present in many operating systems and applications.

In JavaFX, the TreeView control provides a view on to a single tree root, from which it's possible to drill down, recursively, into the item's children. [43] The content of the TreeView are TreeItems, which are nodes that hold a single value. These items, however, don't fire any visual events and cannot be styled, although this behaviour can be obtained using custom cell factories which create TreeCells. Cells in JavaFX are used to render single rows in virtualised controls such as the TreeView. [44] Each cell is associated to a single item and is responsible for rendering it. Since there's the possibility that a TreeView can have extremely large data sets, cells are reused, which means that the item that a cell is associated to in one UI update, may not be the same as the one in the next update.

There are 3 elements needed to display an item in a TreeView: the data model, the TreeItem and the TreeCell. The data model is where all the information for that specific level should be saved, the TreeItem should only implement behaviour methods and the TreeCell should handle the way the item is displayed in the interface. As an example, let's analyse the SIP nodes from the classification plan tree. These nodes have an icon and a simple text value, both change colour depending on the node's selected state, white when the the node is selected and black otherwise. In this case, the TreeCell is what makes the colour change by applying different styles when the selected state changes, the TreeItem contains the icon and get's the value from the data model, which is a SIP object with all the properties, title, content, etc. Whenever the description level is updated, the TreeItem updates its icon by fetching it's name from the SIP and getting the icon associated with that name. Note that the SIP doesn't know which icon corresponds to a description level since that's responsibility of the UI layers above it. To summarize, the data model is a SIP object, the TreeItem get's information from that SIP, in the example the value and icon, and the TreeCell styles the item and defines the way it is presented.

43 https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TreeView.html
44 https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Cell.html

4.9.2 *File explorer algorithm*

Due to the states that an item can be in and the fact that we don't necessarily know the full domain being used (since we don't load the full tree right away to improve performance), an algorithm that assures the file explorer is coherent with the rest of the application was developed. This algorithm uses a collection of paths and its states. Since each path can only have one state at a time, a one-to-one relationship can be used. When an item is added to the file explorer tree, it is also added to this collection. For performance reasons, as stated earlier, the full tree is not visited right after a folder is added to the file explorer, it's only when the SIPs are being created that information of the full tree is accessed.

When getting the state of a path, the collection is checked for associated states. If there are any, the state of the path is the one saved in the collection, but when the path does not exist yet, some more computations are needed to find it. The algorithm starts going up the tree until it finds an ancestor's path that is in the collection. When found, the state of the path is the state of the ancestor.

There are a few rules that were defined to keep the state coherent:

- A folder is ignored only if all of its children are ignored as well.
- A folder is mapped if at least one of its children is mapped and no child has the normal state.
- A folder is normal if at least one of its children has the normal state.
- A state can only be set as mapped or ignored if the previous state was normal. We cannot have ignored to mapped changes or vice-versa.

With these rules in mind, additional steps are needed when adding or updating a path's state, since this can mean that one of its ancestors may change its state. When mapping or ignoring a file a shortcut can be taken by only applying the state to the paths already in the collection (since all the other will infer their state by using the information of their ancestors), but the same cannot be performed on the other way around, when a path is set as normal when it was mapped or ignored. This is because the user performed operations are unknown, therefore it can't be safely deduced that all the children of a path are to be set as normal as well. When removing the mapped state of a set of paths, it must be assured that only those paths are affected (and correctly update the ancestor's state), since there could be other mapped paths that would be affected if the same kind of shortcut was used. As a consequence of this, removing SIPs is a task that takes much longer to perform than the creation process. But as removing previously created SIPs are a much less common task than creating the SIPs, this disadvantage does not have a big impact.

The algorithm starts when the `addPath()` method is called, which receives in argument the path that will be added to the collection and the state that it will be associated with. The heavy work is done by two auxiliary methods: `applySameStateAllChildren()` (see 4.2) and `verifyStateAncestors()`.

This method applies the `state` state argument to all the descendants of `path` where the state is `previousState`. It's useful when we're ignoring (or mapping) a folder and all its Normal descendants need to be set as ignored.

```
private static void applySameStateAllChildren(String path, SourceTreeItemState
    previousState, SourceTreeItemState state) {

  states.put(path, state);
  Map<String, SourceTreeItemState> children = getAllChildren(path);

  for (String child : children.keySet()) {
    if (states.get(child) == previousState) {
      states.put(child, state);
      // update the item
      if (items.containsKey(child)) {
        items.get(child).setState(state);
      }
    }
  }
}
```

Listing 4.2: The method that applies one state to all descendants of a path

The getAllChildren() method (see 4.3) takes advantage of Java 8's Stream API to filter the paths of the collection that start with the path received as parameter. We can use the String's method `startsWith()` because a children path must start with its parent's path. In parallel, if we want to get the direct children only, we must count the number of file separators from the parent and select the paths that start with the parent path **and** their file separators count is the parent's count plus one.

```
private static Map<String, SourceTreeItemState> getAllChildren(String path) {
  return states.entrySet().stream().parallel().filter(p -> p.getKey().startsWith(
      path)) .collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()));
}
```

Listing 4.3: The method that returns all children of a path

The previous snippet of code shows the advantages of using the new Java 8 Stream API over features of previous Java versions. To implement the same logic without the Stream API, we would need more lines of code and possibly have lower performance since we

wouldn't necessarily take advantage of parallelism. The following snippet (see 4.4) shows one of the possible ways this method could be implemented without using Java 8 features.

```
private static Map<String, SourceTreeItemState> getAllChildren(String path) {
  Map<String, SourceTreeItemState> result = new HashMap<>();
  for(String p: states.keySet()){
    if(p.startsWith(path)){
      result.add(p, states.get(p));
    }
  }
  return result;
}
```

Listing 4.4: The getAllChildren method without using Java 8

The `verifyStateAncestors()` method iterates the ancestors of a path and verifies if they have the correct state. It's commonly used after the path received as argument has a change of state. For every path separator ("/" for Unix systems or "ẗor Windows systems) the substring from the start to the current separator is considered an ancestor's path which will be verified. This verification per path implements the rules that keep the states coherent, as described above. If a path doesn't have the correct state after the verification, its state is updated and its ancestors are verified.

There are some occasions where we don't want the algorithm to run but still want to add paths to the collection, for example, when adding a new folder to the file explorer. In this case, all the paths are automatically set as Normal, since the user couldn't have set them in any other state before they were added to the file explorer. This method, which only adds a path to the collection, is much quicker than running the full algorithm.

Before finding the current solution, several more were tested and researched. The first approach was to keep the state of the path in the associated tree item. Every time the state needs to be changed, the item is updated and all its children are notified of the change. For simple cases of state changes, this approach worked, but for complex cases the tree would have an unpredictable behaviour where items didn't properly transition from one state to the other. One of the reasons was that we could not go up in the tree since the item's didn't have a reference to their parents, which meant that we couldn't implement methods like the `verifyStateAncestors()` previously described. Even after adding a reference to the parent of each item, trasversing the full tree was inneficient since we needed to go up and down between item references. Furthermore, we didn't have an easy way to check what state a random path has and needed to (in the worst case) search the full tree to find its corresponding item.

The solution to the above problems was to create a public collection where the state changes are saved. The first option was to have a "stack" system (a list where the last/top

Figure 22: The collection option with the state changes stack. Note that the second and fourth entries negate each other.

items were the last to be added) where each state change would be a new entry of the stack. Inverse entries would negate each other and both entries would be removed from the stack, for efficiency reasons. Every time we needed to know the state of a path, an algorithm would iterate the full list of operations and compute the correct state of the path. In addition to being harder to implement, this algorithm would be increasingly slower has the amount of stack entries grew, so it was discarded. The second option was similar to the previous approach, but instead of a stack of state changes, the collection would be a map where each entry associates a list of timestamped state changes to a path. Since it had the same problems has the first option, it was discarded as well. The third and last option is the one currently implemented and in addition to being much simpler than all the others, it has worked for every case tested, so far.

4.9.3 *Walking the file tree*

The file system is the source of the content that is added to the SIPs, so we need a way to walk it. Starting in version 7, Java added a way to visit files using the `walkFileTree` [45] method and the `FileVisitor` [46] interface. The first is a static method of the `Files` class and receives a starting path as the first argument and a FileVisitor as the second argument. Optionally, we can use a second variation of this method and specify how deep the visitor should go and which options should be taken (for example, following symbolic links). The File visitor interface specifies the required behaviour at key points in the traversal process: when a file is visited, before a directory is accessed, after a directory is accessed, or when a failure occurs. The interface has four methods that correspond to these situations: [47]

- **preVisitDirectory** – Invoked before a directory's entries are visited.
- **postVisitDirectory** – Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
- **visitFile** – Invoked on the file being visited.

45 https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#walkFileTree-java.nio.file.Path-java.nio.file.FileVisitor-
46 https://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html
47 https://docs.oracle.com/javase/tutorial/essential/io/walk.html#filevisitor

- **visitFileFailed** – Invoked when the file cannot be accessed. The specific exception is passed to the method.

In addition to being used to add content to the SIPs, this approach is also used to populate the footer of the file explorer with the number of files, directories and size. Any of these actions can be lengthy and consume most of the available resources, which in result make the application unresponsive for large periods of time. To avoid this problem, the file walking is executed in a different thread from the JavaFX thread. Moreover, by using a separate thread, we can notify the main thread of the progress of the file walking and also cancel the process if needed.

4.9.4 *Creating SIPs*

SIP creation is the main purpose of RODA-in, so most of the effort when developing the application was dedicated to implement it. The process starts with the user choosing which paths will be added to SIP(s), as well as what aggregation and metadata options will be used to mold each SIP. The elements required to start the process are:

- The set of paths from which the file tree walking begins
- The identifier of the aggregation option
- The identifier of the metadata option
- The path of the metadata file, if applicable
- A string with the arguments of the metadata option. At the moment, this can be either the template's name or the glob used to get the filtered metadata paths.
- The template version, if applicable
- The type/format of metadata
- The ID of the node where the SIPs will be appended

This set of elements is used to create a **Rule** object. A Rule is an object that makes the bridge between the User Interface and the SIP creation thread. To create the SIPs, it first instantiates a WalkFileTree, providing, as argument, an object of the SipPreviewCreator class. At each of the events described in the previous section, the file walking calls the appropriate method of the creator, which handles it in the required way to build the SIP. For example, when a file is found and the `visitFile` event is triggered, the `visitFile()` method of the creator is called. Should the creator generate a SIP for every file it founds, the `visitFile()` method will generate and save a new SIP using the discovered file.

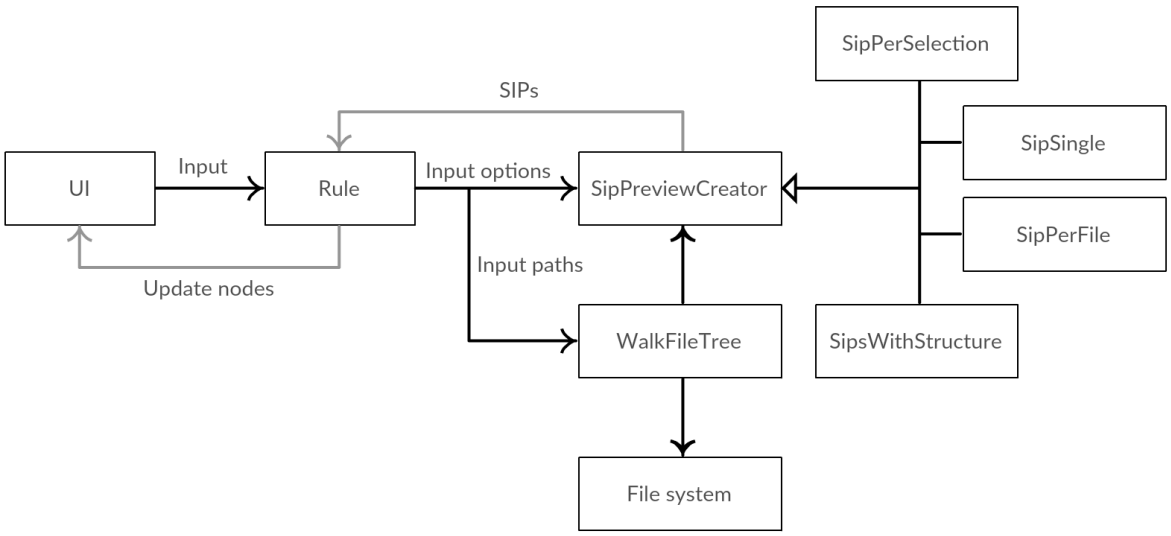


Figure 23: An overview of the SIP creation process.

To start the creation process, the Rule looks at the identifier of the aggregation option and instantiates the respective creator. The SIP creators are instances of the **SipPreviewCreator** class, or of any class that extends it. Although the `SipPreviewCreator` is a perfectly valid creator, it does not create any SIPs and is used mainly as an extendable class that offers the common variables and methods of all the particular creators. For example, metadata association is mostly the same for all creators, so its methods and variables are declared there. At the moment, there are 4 creators in RODA-in: `SipPerSelection`, which creates one SIP for every selected file/folder of the file explorer; `SipSingle`, which creates a single SIP with all of the selected paths; `SipPerFile`, which walks the file tree and creates one SIP for each file it finds; `SipsWithStructure`, which creates new description nodes in addition to SIPs.

The basic behaviour of each of these objects can be described in four steps: preparation, data aggregation, metadata association, preview object creation.

Preparation

The preparation phase is the first step of SIP creation. It's crucial to some of the association options to perform a preparation task, in order to ensure the correct behaviour.

The aggregation options `SipPerFile` and `SipWithStructure` require the full tree to be walked before any SIP can be created. This is due to a shortcoming of the paths algorithm, which would mark the full folder as *mapped* as soon as the first descendant is marked as mapped.

A simple example is a folder with 10 files (let's call it "Folder A"), which has never been opened in the file explorer. The paths collection has no information about which files or how many descendants this folder has, the only available information is that Folder A has the *normal* state. As soon as the first SIP is created using the first descendant of that folder, the file of the SIP content is marked as mapped. As far as the paths collection knows, this is the only descendant of the folder, since it's the only path that has been added to the collection as a descendent of Folder A. Therefore, it deduces that the folder must also be marked as mapped, since all its descendants are mapped as well. As the tree file walking continues and new descendants are found, the inherited state is no longer *normal*, it's *mapped*. As the path reaches the filtering stage [48], it will be filtered out of the SIP creation due not having a normal state. Concluding, due to the first path being marked as mapped the output will be a single SIP, instead of the expected 10 SIPs.

This problem is fixed in the preparation stage by walking the full file tree of the selected paths and adding its descendants to the paths collection as normal. This way, the collection has complete knowledge of the content of each folder and can deduce correctly its state. As it's an additional step which requires I/O access to the disk, it may affect the performance of the aggregation when using the highlighted options.

One more case where the preparation phase is useful is to improve the performance of the "Load from a different directory" metadata association option. This option requires the full list of paths of the user-selected folder, in order to find the item that corresponds to each SIP. The first implementation called the `File.list()` method [49] every time a SIP requested metadata association. This means that for M SIPs, the File.list() method would be called M times with O(n) complexity when getting the correct path from the resulting array. I found that this metadata option had the worst performance of all and researched ways of improving it.

The solution was to use the NIO.2 Java API [50] to replace the File.list() calls, in addition to creating a map with the metadata file names in the preparation phase. Each file name can be split in two parts, the extension (.jpg, .doc, .xml, etc) and the basename. The new map associates a List of Strings, which are file names with extension, to a String with the common basename of the List. For example:

```
dissertation -> [dissertation.pdf, dissertation.tex, dissertation.pdf]
```

With this approach, when a SIP requests metadata using this option, the algorithm only needs to directly get the list of values from the map. These modifications removed the need to perform the costly `File.list()` for each SIP and changed the complexity of the following action from O(n) (when iterating the files array) to O(1).

48 The filtering stage is part of the data aggregation phase which will be described in the next section
49 https://docs.oracle.com/javase/7/docs/api/java/io/File.html#list()
50 https://docs.oracle.com/javase/7/docs/api/java/nio/file/package-summary.html

Data aggregation

The data aggregation phase uses the files and folders discovered by the WalkFileTree object to aggregate the information in packages. It does so with the help of an additional object, `TreeNode`, which simply holds a path and the list of its direct children. Before a path can be processed any further, it must first pass through a series of filters. This is how we avoid creating SIPs with ignored or mapped paths.

TreeNode creation uses a deque [51] of objects to help create the tree and a Set to save the top files/folders of each SIP. The strategy on how to create and organize these nodes is unique to each creator.

The **SipPerFile** creator is the simplest of all. Since it's used to create a SIP for every file, it creates a TreeNode for every file it finds. **SipSingle** and **SipPerSelection** create a new TreeNode before visiting each directory and add them to the deque. Every file discovered is added as a TreeNode to the top folder of the deque, since this is its parent folder, or to the top items Set if there's no open folder in the deque. When the postVisitDirectory() method is called, the TreeNode at the top of the deque is closed and added to the next node of the deque, if it exists, or to the top files Set otherwise. This effectively creates a tree of nodes since the top elements of the deque are children of the bottom elements, e.g., the node on position 3 is a child of the node on position 2, which in turn is a child of the node on position 1. The **SipWithStructure** creator is more complex, since it needs to create the full tree structure and make decisions on which TreeNodes are added to SIPs and which are used to create Series. This last creator uses a bottom-up approach to make these decisions, since the content of the nodes on the bottom influence what the nodes above will be once they're created.

Metadata association

Once the content of the SIP is defined, we can add the metadata. The metadata association phase adds metadata to the SIPs and is strongly linked with the preview object creation phase. Every SIP may have multiple metadata objects, which save the metadata association type and the necessary data to create the metadata file (such as paths, template name and version, raw content, etc.) once it's requested.

In order to reduce memory usage, the full content of the metadata files is not loaded in memory as soon as the SIP preview is created. Instead, we keep just the necessary information to create these files and only load them if the user makes a request for it. SIPs which have been inspected using the user interface have their metadata loaded in memory, while the SIPs which haven't been inspected yet don't.

51 https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

The templating metadata option uses information available in the SIP to fill the gaps of a pre-made metadata file. The process on how this is achieved is described in the next subsection. The second option, metadata from a file, simply adds an existing path as a metadata file, without further processing. The third option, metadata from each directory, uses a `PathMatcher` to filter paths that match a certain glob (see 4.5). This option can produce more than one metadata file since a glob is able to match with multiple paths.

```
PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:" +
    userInputGlob);
File[] foundFiles = dir.listFiles((dir1, name) -> matcher.matches(Paths.get(name)
    ));
```

Listing 4.5: Listing the files that match a pattern

The last option, metadata from a different directory, uses the map created in the preparation phase to retrieve files that share the same basename as the one in the SIP. Since the path of the directory chosen by the user can also be used to create the SIPs, we need to filter out the files included in the SIP. For example, a directory with files roda-in.txt, roda-in.pdf and roda-in.xml is used to create SIPs but is also added as the source of the metadata files. Only the PDF file was added to the SIP since the other two were ignored in the data aggregation phase. When the metadata association phase is reached and the algorithm retrieves the paths with the basename "roda-in" from the map, three paths are returned, but since the PDF was added to the SIP, its path is removed from the list of metadata files.

Preview object creation

Creating the preview objects is the last step of the SIP creation. While in some creators this step happens after the data aggregation phase, in others the preview creation is performed at the same time as the data is aggregated in nodes.

The SipSingle creator creates a single SIP containing the full tree of the selected paths, so it needs to wait for the data aggregation to end before it can create the SIP preview. On the same vein, the SipsWithStructure creates a temporary node tree and only creates the SIPs and other description levels after the full file tree has been walked.

On the other hand, the SipPerFile creator can instatiate a SIP object every time a file is found, effectively creating the SIPs as the data aggregation phase is performed. The SipPerSelection creator also creates SIP previews as the items are discovered, provided that the file or folder being walked was one of the initially selected paths by the user.

Once all SIP previews are created, the full list is sent back to the Rule, which establishes the bridge between the creator and the UI. The preview objects are wrapped in a SipPreviewNode, which are added to the classification plan tree to later be exported or inspected.

4.9.5 *The templating system and forms*

The templating system was implemented using the Handlebars templating engine, specifically the Handlebars.java Java port. [52] The objective is clear: we want a system that is extendible enough to support multiple personalization options, easy to use and doesn't clutter the template so it stays easy to read. After some research and brain storming, I had three options to choose from:

1. Comments - Create a Domain Specific Language (DSL) and look for it in the template's comments. This approach has the advantage of enabling the user to organize the information needed to create the forms in a single place of the template, for example, the beginning. On the other hand, it requires a specific parser and adds more chances of errors. An example of a valid comment is:

   ```
   <!-- $${ id="title" type="text" required=true order=4 }$$ -->
   ```

   where `$${}$$` is what differentiates a comment as information that needs to be extracted from a standard comment.

2. Separate file - Another option is to create a separate file where all the information to create the form is stored. This file could be linked to the template file by sharing its name and using a different extension, or by a specific field in the template. This approach separates the concepts of template and form by preventing the information to be shared. Like the previous option, this also requires a DSL to structure the information and a custom parser to extract it.

3. Tags - Mix the information needed to create the templates in the Handlebars tags. This approach is inspired by how the web framework Ember.js [53] handles its templates. In addition to the normal tags, its possible to add additional information to tell Ember how to create the resulting HTML or how to bind to the underlying model. This way, we can wrap the necessary information to create each field of the form in each tag.

Due to being easier to use, easier to implement and the one that would cause the least confusion of the three, the chosen approach was number 3. How this is implemented depends on whether the Handlebars parser has support for additional options and whether we can create tags beyond the ones that the library supports by default.

The "tags" in Handlebars are called **helpers** and specify functionalities of the engine and how values should be applied to the template. Some of the supported helpers are:

EACH Used to define custom iterators where an action is executed for each item.

52 https://github.com/jknack/handlebars.java
53 http://emberjs.com/

IF Used to evaluate condition statements. Usually followed by the **else** helper.

UNLESS The same as the **if** helper, but the action is only executed when the condition is false.

PARTIAL A partial block works as a template that can be included into other templates. Recursive partials are possible.

BLOCK Used to provide template inheritance. If the template (or partial) that needs to be loaded doesn't exist, the helper body is used.

The Javascript and Java implementation of Handlebars have methods to register new helpers, which require a name and a callback function. This opens the possibility of defining specific helpers for creating the fields of the form. We can define a single helper, for example **input**, and extend it to contain an ID attribute, a name attribute, etc. Or we could define multiple helpers, one for each of the available field types, and expand it. There are many possibilities if we want to register fixed helpers, but this was not the approach taken.

The Java implementation of Handlebars has a functionality that isn't available in the Javascript implementation, the `registerHelperMissing()` method. By default, Handlebars.java throws an `java.lang.IllegalArgumentException()` if a helper cannot be resolved. Using this method we can override the default behaviour, catch all the unresolved helpers and obtain their information. Now we're not limited by the hard-coded helpers we decided to implement, instead we can use whatever helper name we want and even use it as the identifier for that field in the form.

Instead of using the standard approach to replace helpers with values in one pass of the template, we need to make a pre-pass to extract the helper names and options so that we have the information to create the form. The first pass is achieved using the `registerHelperMissing()` method (see 4.6). The MetadataValue class instantiates objects with an id and a map of options and each object represents a future field in the form. We only add new MetadataValue objects when the current helper has a context since this way we can avoid adding the pre-defined helpers (such as if, each, unless, etc.) to the map and wrongly create fields with them. The second part of the conditional statement is used to avoid multiple tags from being added to the map, we only add the first one that appears and ignore the rest. Fortunately, the Handlebars parser already parses the options and adds them to a map, which we can access using `options.hash`. The last two lines of the try statement compile the raw template and apply an empty map, since we don't replace any values in the pre-pass.

```
public static TreeSet<MetadataValue> createSet(String content) {
  TreeSet<MetadataValue> result = new TreeSet<>();
  Set<String> addedTags = new HashSet<>();
  Handlebars handlebars = new Handlebars();
```

```
  Template template;
  try {
    handlebars.helpers().clear();
    handlebars.registerHelperMissing((context, options) -> {
      String tagID = options.helperName;
      if (context != null && !addedTags.contains(tagID)) {
        result.add(new MetadataValue(tagID, options.hash));
        addedTags.add(tagID);
      }
      return options.fn();
    });
    template = handlebars.compileInline(content);
    template.apply(new HashMap<>());
  } catch (IOException e) {
    e.printStackTrace();
  }
  return result;
}
```

Listing 4.6: The method that creates the set of variables from the template

Due to how each helper is created, the standard way of using the Handlebars engine to replace values cannot be used. Take the following as an example:

```
<title>{{title order=2 auto-generate='title'}}</title>
```

Using the standard method, if we input a Map which associates the key **title** to the value **Creating SIPs with RODA-in**, the expected output would be:

```
<title>Creating SIPs with RODA-in</title>
```

Unfortunately, this doesn't happen because the engine parses **title** as a helper instead of a replaceable expression. To overcome this hurdle, we need to once again use the `registerHelperMissing()` method, where we check if each helper is present in the values map. The values map associates a helper name with the value that it should be replaced with. Whenever one of these helpers is found, the associated value is returned so that the engine replaces the full handlebars expression – {{ ... }} – with the correct value.

5

EVALUATION

This chapter presents the testimony of specialists in the area of digital preservation, namely archivists of the Portuguese *Direção-Geral do Livro, dos Arquivos e das Bibliotecas* (DGLAB) [1], which serve to validate the approach taken during this work. The development process was performed under the constant supervision of archivists from DGLAB, particularly Francisco Barbedo, Lucília Runa, Ana Rodrigues and Mário Santana, and the E-ARK project, specially Szatucsek Zoltán from the National Archives of Hungary.

Lucília Runa has been an archivist since 1990 and currently works in the Innovation Services and Electronic Administration Direction of DGLAB. Since 1999 most of her professional activity has been in the archivist description area, building and applying standards, description quality control and access to information through national and international portals.

The following is a used case described by Lucília where RODA-in 2.0 was used to create SIPs.

> "We used RODA-in 2.0 to create SIPs for approximately 50 files of different formats, ranging from 30 KB text files to 500 MB video files. There was no difficulty when creating the SIPs and for the present case the exportation was quick."

The formats and approximate dimensions of the files used are as follows:

- Text – PDF (ca. 1500 KB), XML (ca. 30 KB), DOC (ca. 60 KB)
- Spreadsheet – XLS (ca. 1500 KB)
- Audio – WMA (ca. 4000 KB)
- Video – AVI, MP4, MPG, WMV (ca. 500000 KB)
- Images – TIFF, JPEG (ca. 900 KB)

1 http://dglab.gov.pt/

- PPT (ca. 1500 KB)

Furthermore, Lucília was asked to evaluate the performance of RODA-in when adding metadata to the packages:

> "Generally we didn't detect any problems when creating or associating metadata. The multiple options offered eased and streamlined the process of creation and association of metadata and adapted well to different types of situations and requirements.
>
> The metadata was frequently created by RODA-in. Often completed or altered using the in-app forms, individually, or in batch. However, for many of the archived files the metadata already existed in an XML format which was easily associated to the material using the several options offered by RODA-in, specifically *Load from a single file*, *Load from one directory* and *Load from each directory*."

In addition to this case, Lucília and the DGLAB team presented RODA-in to different entities and tested the application using files produced by those entities. These tests are crucial since they use RODA-in in real world scenarios where the files have varied structures/folder hierarchies, sizes and formats. The entities that tested RODA-in with the help of DGLAB were:

CINEMATECA PORTUGUESA - MUSEU DO CINEMA

National organization overseen by the Portuguese Culture Minister with the mission of safeguarding and disseminating the Portuguese film heritage.[2]

TEATRO NACIONAL D. MARIA II

The National Theatre D. Maria II is a public entity which aims to enhance the relationship of the theatre with the country and provide content creators a privileged space to display their talent and work.[3]

BIBLIOTECA DE ARTE, DA FUNDAÇÃO CALOUSTE GULBENKIAN

The Calouste Gulbenkian Foundation is a Portuguese private foundation which supports a myriad of cultural activities. The Art Library is principally focused on visual arts and architecture.[4]

COMBOIOS DE PORTUGAL

CP - Comboios de Portugal is a Portuguese company which operates freight and passenger trains.[5]

2 http://www.cinemateca.pt/
3 http://www.teatro-dmaria.pt/
4 https://gulbenkian.pt/biblioteca-arte/en/
5 http://www.cp.pt/en/

Following the tests performed by these entities, Lucília notes:

> "Generally, all the entities considered RODA-in very friendly, intuitive and that it adapted to their needs, having shown a great deal of interest in using it."

6

CONCLUSIONS AND FUTURE WORK

The goal was to develop an offline SIP creation tool to tackle the problem of mass data archival. RODA-in allows the producer to create thousands of SIPs with gigabytes of data in few clicks, without overlooking the needs of more advanced users. Since the file system is the source of the data and the user must have full control over it, I developed a file explorer that supports 3 states: normal, ignored and mapped. The classification scheme can be imported from a previously saved plan or created directly in the application.

To create SIPs there are 4 aggregation rules that should cover the needs of most users: one SIP for each selected file or folder; one SIP containing all selected files and/or folders; one SIP for each file under the selected folder(s); create classification scheme and SIPs based on folder structure. In the same way, metadata can be associated to the created SIPs automatically, using 4 association options: metadata based on a template; same metadata in every SIP; load metadata from the data folder; load metadata from a separate folder.

The resulting SIPs can be edited and enriched using metadata forms based on templates, rearranging and removing files from the data section of the SIP or adding documentation. Additionally, the metadata can be modified in batch by selecting multiple items from the classification plan. The last step is to export the SIPs to an output folder so that they can be later added to a repository. Currently, RODA-in supports two SIP export formats: BagIt and the E-ARK SIP.

RODA-in is available in several languages and new idioms are being added constantly due to the efforts of its users. The application has been piloted by the Portuguese *Direção-Geral do Livro, dos Arquivos e das Bibliotecas* (DGLAB) [1] and its producers, and by the E-ARK project associations, such as the Archives of the Republic of Slovenia or The National Archives of Hungary.[2] The feedback has been extremely positive, several features have been implemented and bugs have been fixed due to the tests performed by these organizations. Additionally, this work has been presented to the digital curation community in *Encontro de Curadoria Digital* organized by the *Universidade Nova de Lisboa* [3].

1 http://dglab.gov.pt/
2 http://www.eark-project.com/partners-1
3 http://curadoriadigital.fct.pt/

In the future RODA-in may support transfer projects which would enable the user to have projects for SIP creation. The producer would be able to configure the application in a project by project basis, modifying metadata values, templates and types at will. Additionally, the full state of the application would be preserved, which would be useful for complex SIP creations that can last days. Finally, it would support the importation of a report created by the repository, for example RODA, that would notify the user of which SIPs were successfully ingested and which were not, complete with the error details. The producer would then be able to correct the failing SIPs and retry to ingest them.

The templating system will possibly need an overhaul. For small and simple templates, the current system works perfectly, but for larger files with more variables, the template becomes cluttered and difficult to read. A new proposal is to move all the field declarations to the top of the page, and simply use the variable name inside brackets where we want to place the values from the form. This would clean the template making it easier to read and also facilitate the edition of the fields.

Currently RODA-in can only export SIPs, however, a constantly requested feature is the ability of exporting the classification levels as well. These new items can then be imported by the repository to recreate the tree as it was created in RODA-in. Furthermore, the ability to create *Update SIPs* is also crucial to some producers. The purpose of these SIPs is to update the content (descriptive metadata, documentation and representations), of SIPs that were ingested in the past.

BIBLIOGRAPHY

Andy Boyko, J Kunze, J Littman, L Madden, and B Vargas. The bagit file packaging format (vo. 97). *Washington DC*, 2011.

CCSDS. Reference Model for an Open Archival Information System (OAIS). Magenta book. Technical report, 2012. URL http://public.ccsds.org/publications/archive/650x0m2.pdf.

CCSDS. Producer-Archive Interface Specification. Blue book. Technical report, 2014. URL http://public.ccsds.org/publications/archive/651x1b1.pdf.

Lois Mai Chan and Marcia Lei Zeng. Metadata interoperability and standardization – a study of methodology part i. In *D-Lib Magazine*.

D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

Luís Francisco Faria. *Automated Watch for Digital Preservation*. PhD thesis, Universidade do Minho, 2015.

Miguel Ferreira. *Introdução à preservação digital – Conceitos, estratégias e actuais consensos*. Universidade do Minho. Escola de Engenharia, 2006.

ISO. Space data and information transfer systems – open archival information system (oais) – reference model. iso 14721, 2012a.

ISO. Space data and information transfer systems – audit and certification of trustworthy digital repositories. iso 16363, 2012b.

Tarvo Kärberg, Karin Oolu, Piret Randmäe, Kathrine Hougaard Edsen Johansen, Alex Thirifays, Boris Domajnko, Janet Delve, and David Anderson. D3.1 - report on available best practices. Public deliverable, E-ARK, September 2014.

Tarvo Kärberg, Karin Bredenberg, Björn Skog, Anders Bo Nielsen, Kathrine Hougaard Edsen Johansen, Hélder Silva, Gregor Zavrsnik, Levente Szilágyi, and Phillip Mike Tømmerholt. D3.3 - e-ark sip pilot specification (revision of d3.2, main part of the d3.3. Public deliverable, E-ARK, February 2016.

Anne R Kenney, Nancy Y McGovern, R Entlich, WR Kehoe, E Olsen, E Buckley, and C DeMello. Digital preservation management: implementing short-term strategies for long-term problems. In *online tutorial and workshop, Cornell University Library*, 2003.

Brian Lavoie and Richard Gartner. Preservation metadata (2nd edition). Technical report, Digital Preservation Coalition, 2013.

Jean-Louis Letouzey. The sqale method - definition document. Technical report, January 2012. URL http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf.

PREMIS Editorial Committee. Data Dictionary for Preservation Metadata: PREMIS version 3.0. Technical Report. Technical report, Library of Congress, 2015.

R. Rivest. The md5 message-digest algorithm, 1992.

K. Thibodeau. Overview of technological approaches to digital preservation and challenges in coming years. *The state of digital preservation: an international perspective*, 2002.

University of Illinois. Best practices for creating digital collections. Technical report, University of Illinois at Urbana-Champaign, 2010.

Eloi Juniti Yamaoka and Fernando Ostuni Gauthier. Objetos digitais: em busca da precisão conceitual. 18(2):77–97, 2013. ISSN 1981-8920. doi: 10.5433/1981-8920.2013v18n2p77. URL http://www.uel.br/revistas/uel/index.php/informacao/article/view/16162.