UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

# CS/IT Honours Project
# Final Paper 2022

**Title: Mr**


**Author: Craig Stevenson**


**Project Abbreviation: Arch2**


**Supervisor(s): Hussein Suleman**


| **Category** | *Min* | *Max* | Chosen |
|---|---|---|---|
| Requirement Analysis and Design | *0* | *20* | 20 |
| Theoretical Analysis | *0* | *25* | |
| Experiment Design and Execution | *0* | *20* | |
| System Development and Implementation | *0* | *20* | 20 |
| Results, Findings and Conclusions | *10* | *20* | 10 |
| Aim Formulation and Background Work | *10* | *15* | 10 |
| Quality of Paper Writing and Presentation | *10* | | 10 |
| Quality of Deliverables | *10* | | 10 |
| Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*) | *0* | *10* | |
| **Total marks** | | **80** | |

# Ingestion Tool For an Archive of Archives

Craig Stevenson
Computer Science
University of Cape Town
Cape Town
Stvcra005@myuct.ac.za

## ABSTRACT

Archives are meant to be a safe haven for information however many archives can simply disappear due to a variety of reasons such as lack of funding, world events such as natural disasters or wars. In addition changes to archives over time are not documented anyway where. This a problem simply because it takes valuable information out of circulation and is often never returned. In order to solve this problem, an archive that is able to store other archives called the *Archive of Archives* will be created. This archive will be comprised of four components: UI, Repository, Ingestor and Web-scraper. This paper presents a part of the solution of this problem which is the design of the Ingestor tool. The objective of this tool is to ingest and version a web-scraped archive into the repository without any data loss occurring.

## CCS CONCEPTS

• Information Retrieval • Digital Archiving • Data Scraping

## KEYWORDS

Digital Archiving, SimpleDL, archive of archives, versioning

## 1 Introduction

The internet is home to a vast amount of knowledge, much of which we do not have control over. Archives exist for the purpose to control and preserve important collections of information such that they will be available for future generations. Archives however are tied to organizations who need to upkeep the servers they run off of. This means that at any given time a knowledge source, one depends on, may suddenly vanish. Reasons for this include wars, natural disasters or simply funding for a digital archive running out. In addition the creation archives to begin with can be a difficult task since it requires a lot of technical skills and proficiency in the use of digital library toolkits such as Eprints and Dpsace. This will not always be suited for lower resource environments.

In order to combat this a digital library that is able to store other digital libraries and version them, the A*rhive of Archives,* will be created. The *Archive of Archives,* as seen in figure 1, will be comprised of four main parts: (1) UI; (2) SimpleDL Repository, which is a flat file format repository with indexing and search functions; (3) Ingestor and Metadata Scraper and (4) Archive Collector, which is a webscraper.

This *Archive of Archives* ultimately aims to: (1) digital resources will be preserved in the long term, (2) changes to digital resources will be recorded and stored to produce versions of those digital resources, (3) we shall provide offline functionality for the digital library such that content will be preserved through crashes, as well as providing areas with poor network connections, (4) preserve the look and feel of the archives scraped, by scraping the actual HTML pages, (5) Act as a simple easy to understand toolkit that many can use to create their own archive of archives.

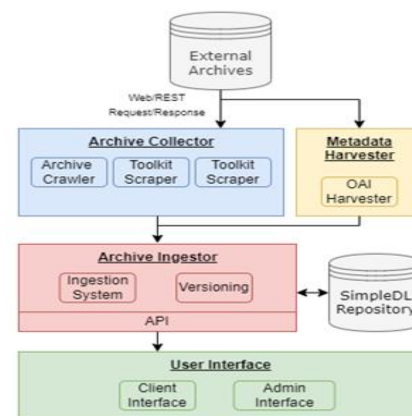


**Figure 1: Architecture Diagram showing all the core parts of the archive of archives and how they interface with one another**

This paper focusses on the software engineering process used to create the Ingestor and Metadata Scraper parts. This part's overarching responsibility is to: (1) Scrape the metadata for all the digital objects stored in a specified archive and (2) Ingest the output of the *Archive Collector*(See figure 1), without data loss, into the SimpleDL repository so that is stored correctly(with versioning) and the digital objects can be indexed.

This paper details, from a software engineering perspective, how this Ingestor system was created. The paper is structured as follows: Section 2: Related/Background work, Section 3: Requirements analysis design, Section 4: Software development and implementation, Section 5: Test methods and results and discussion, section 6: conclusions, section 7: Future work, Section 8: References, Section 9: Supplementary information/appendix

## 2 Background work

### 2.1 Repository architecture

The repository is the key component of a digital library since all services or layers will have to interact with it. Hence the organization and storage of data is key. The core requirements a digital repository needs to have are: Long Term Preservation / Access to the repository's content, well organized Meta-data, Interoperability, Security/ User Certification and organization of digital objects[1]. Metadata is one of the most important pieces of information in a digital library and hence how it is stored in a repository is a key consideration when designing a digital repository. There are three approaches to its storage: relational biased solution, Documentbiased solution and unbiased solution.

#### 2.1.1 Approach One – a relational biased solution

In this solution Objects and Metadata are stored in a relational multi-media, database. Files are stored as BLOBS and an ER schema will show how to connect these tables in the DB to tables holding meta data.

#### 2.1.2 Approach two – Document biased solution

Files are stored in database(with pointers/labels) and metadata stored in an XML document repository. Business logic is hence needed to associate the two components, requiring a lot of code. In addition unique ids are needed in both xml and actual files.

#### 2.1.3 Approach three – unbiased solution.

Use a relational DB with native XML support. Metadata stored separately from the binary files, in XML documents that follow their own schema. This approach is the most common approach [1]. This approach has several benefits over the above two approaches. With regard to approach (2.2) this approach fixes the problem of needing business logic to link the metadata and the binary data, due to both of them now being managed by one data base management system. This approach is also a direct improvement to approach (2.1) since the metadata is stored separately. This allows for an easy way to contribute to the OAI and allows for metadata about an object to be preserved even if the actual object is removed somehow

## 2.2 Toolkits for building digital libraires

Digital libraries are very complex structures with many components, independent of which architecture you use. In order to simplify the creation of digital libraries, toolkits can be used. The toolkits that will be discussed here are the SimpleDL toolkit, FEDORA toolkit and Dspace

#### 2.2.1 SimpleDL

SimpleDL is a tool for creating pre-generated digital libraries in a low resource environment[2]. SimpleDL allows for the long term preservation of data and retains data through network failures or computer system crashes. In addition, simpleDL allows for the easy migration of data. This toolkit has several benefits for the archive that we are building. Most notably the fact that it has support for offline functionality, which is something our archive needs to have. A disadvantage of simpleDL is that our archive will store a lot of digital objects and simpleDL can only return results in a feasible amount of time for up to 100 000 items, of which our digital repository may contain more. Due to its minimalist design this toolkit is very easy to extend to add other features as well as maintain making it a suitable choice for this project

#### 2.2.2 Dspace

Dspace is another tool used for creating, normally, institutional digital libraries[3]. This tool provides many core aspects a digital library requires robust repository architecture, search and browse functions, web user interface, ingesting functionality and OAI support, all in one toolkit. Dspace orders members of the community that will use the digital library in a hierarchical fashion to determine their interaction permissions (namely addition and deletion permissions)[4]. Dspace follows a layered architecture approach, consisting of three layers: Application Layer, Business logic layer and storage layer. Dspace is considered to be the most popular tool for creating digital libraries[5]

#### 2.2.3 FEDORA

Another common tool that specializes in assisting in digital repository design is FEDORA. This service architecture consists of three layers: Web Services Exposure Layer, the Core Subsystem Layer and the Storage Layer[6]. This tool is useful since our digital library needs to be able to store complex objects(other archives). This tools main features include: XML submission and storage, Parameterized disseminators, Access Control and Authentication, Default Disseminator, Searching, OAI Metadata Harvesting and Batch Utility. Features such as versioning, which is an important requirement for our archive, will be made available in future updates. Fedora has four main use cases: Fedora "out-of-the-box", A digital asset management system, A digital library for a research university and Fedora for distributed content objects. In practice a typical implementation uses a blend of all four of these use-cases. [7] presents a case study of where FEDORA was used to create an extensible digital repository. The digital repository in this paper has a similar goal to ours since this repository was also used to store already made collections of digital objects. The Fedora architecture allowed them to have: support for heterogeneous data types; (2) accommodation of new types as they emerge; (3) aggregation of mixed, possibly distributed, data into complex objects; (4) the ability to specify multiple content disseminations of these objects; and (5) the ability to associate rights management schemes with these disseminations

SimpleDL has many differences with Dspace and FEDORA. This toolkit is designed to create libraries that are a lot smaller in scale to libraries created with DSpace and FEDORA. SimpleDL adopts a more lightweight approach to developing digital libraries, so it wont have all the features Dspace and FEDORA have. What

SimpleDL does that DSpace and FEDORA cant do, is that it provides offline functionality and hence reduces the dependency on remote servers. SimpleDL is also a lot simpler to maintain and build since there are less moving parts. One notable advantage simpleDL has over FEDORA is that simpleDL comes with user interface and search engine features. If one uses FEDORA these have to be integrated with it[13]. Like Dspace simpleDL also allows the feature for user profiling allowing one to set permissions of users. Of course Dspace and FEDORA have the advantage of doing a lot of what SimpleDL can do but at a much larger scale. Examples being, that libraries made using these tools, store more data and access data faster than simpleDL libraries and the storage of more complex digital objects can be accommodated. FEDORA and Dspace are both tools that are used in creation of large institutional digital libraries hence they do share some similarities. The tools are compared in the table below.

| Comparison points | Dspace | FEDORA |
|---|---|---|
| 1. OAI support | Yes | Yes |
| 2. UI and search feature | Yes | Needs to be integrated with software that performs that function |
| 3. Architecture | Layered: Application, Business logic and storage layer | Layered: Webservices, core subsystem and storage layer |
| 4. Storage of digital content | Stored as items which are made up of Bundles, Bitstreams and Bitstream formats all of which make up different representation of the file | Stored as objects with mappings to different representations of that objects(for example file formats) |
| 5. File formats supported | Can store any type | Can store any type |
| 6. Complexity | Used primarily to create institutional wide digital libraries | Can be used to design solutions for simpler problems not just for institutional digital library design. For example it can just be for storage and management of simple content objects(e.g. Jpegs, pdfs) |

## 2.3 Offline Archiving approaches

### 2.3.1 Approach One

In this first approach[8] Ajax is used as the tool to create many client side digital library services. One such example is an inbrowser query system. All data is indexed and the inverted files as well as the mapping of id to actual name are stored in XML files. Each inverted file contains part of a document id to link documents of its type, allowing for simple queries.

### 2.3.2 Approach Two

The second approach uses an extended Boolean model, and was designed based off of typical information retrieval policies described in Managing Gigabytes[9]. It uses two applications: create index.pl and search.js. search.js simply builds the indices needed by producing lists of inverted indices for each field. Search.js uses that index to locate the item and display it. This search engine has good performance with the most complicated search taking less than half a second to complete for collections of 32000 items[9].

### 2.3.3 Approach Three

Another approach that allows for offline functionality would be to use the Greenstone architecture. This approach involves indexing collections and then distributing them on a CDROM[9]. Users use the service by first selecting a collection they want to use then, use browsing terms to further filter the results and then can use search terms to find individual words or phrases that occur in selected parts of the document[10]. At a high level the system works by organizing the data into collections, each of which have five directories(import, GML, indices of the collection, building information and support files(e.g., configuration files). When new additions to collections are made, importing occurs, in which source material is converted to GML(Greenstone markup language), which includes any metadata that comes with the document. The building step then occurs; which index the data[10].

All three of these techniques are very suitable to design digital libraries or services of digital libraries. Ajax is a more flexible approach compared to Greenstone since Greenstone requires the installation of their own operating system and Ajax needs just use the technology built into browsers. Ajax technology as mentioned earlier is used to build individual services that can be integrated into digital library systems. This is advantageous over Greenstone since Greenstone is a full package tool and cannot be easily integrated with other client side services. Searching is an important service in digital libraries, each of which the above approaches are able to perform. These search engines have been evaluated in different ways. The Ajax based search engine has been used in the Bleek and Lloyd collection[8]. The search engine used in approach two has been evaluated through a number of experiments ultimately theorizing that this approach can be used on collections of 100 000 items feasibly[9]. Greenstone's search service has been long established and is able to operate on collections that have several thousand to millions of records[10]. While it seems Greenstone is the best option to choose, all of its features are not necessary when designing smaller repositories. Approach one and two are hence advantageous to use in the design of smaller repositories.

# 3 Requirement Analysis and Design

## 3.1 Initial Requirements

This tool forms part of a larger digital library toolkit - the archive of archives, hence most requirements of the system are based on the needs of the other components of the system. These components are: The UI, Simpledl toolkit and the Archive Collector. The main techniques employed in finding the requirements were: (1) Meetings, these included meetings with the project supervisor as well as meeting with other group members developing separate parts of the system, (2) Literature analysis, this included reading documentation about SimpeDL as well information about metadata standards, (3)Presentation feedback, this involved presenting the project proposal to computer science staff as well to the digital library research group at UCT.

The requirements for this tool can hence be summarized into functional and non-functional requirements as follows:

**3.1.1 Functional Requirements**

3.1.1.1 *Data ingestion.* The Archive Collector produces scraped archives that need to be ingested. The ingestor will use the metadata scraper to acquire metadata for each digital object and convert that into a metadata format that SimpleDL can index. The ingestor will also generate metadata for the scraped archive website for SimpleDL to index.

3.1.1.2 *Versioning.* One of the most important aspects of an archive of archives is being able to track changes to an archive over time. This feature detects, when an archive is being scraped, whether or not it exists already in SimpleDL and up to what version. This new archive will then be saved under what version it is detected to be and metadata will be produced to reflect this. Information, including how many items were added or removed and how many items were modified, will be saved.

3.1.1.3 *Ingestion report generation.* After all digital objects from the scraped archive has been ingested into SimpleDL, a report needs to be generated to document how many digital objects were not ingested due to them not being present in the scraped archive. Handles for the digital objects not ingested will be included in the report

3.1.1.4 *API for UI.* Allows user interface to interact with the scrapers to relay instructions on what archives to scrape and when to do so as well as removal of certain archives.

3.1.1.5 *Metadata scraping.* In order to ingest items from a scraped archive the metadata for each of those items is required. Therefore- this tool needs to scrape all the metadata from the archive that is to be scraped.

**3.1.2 Non-Functional Requirements**

3.1.2.1 *Scalability.* The tool needs to able to ingest large archives without crashing or taking too long to complete.

3.1.2.2 *Compatibility.* The tool needs to be able to run on MS - Windows and Linux operating systems as well as work with SimpleDL.
3.1.2.3 *Reliability.* The ingestor needs to be able to ingest digital objects from a scraped archive at anytime with 100% accuracy.

## 3.1.1 System Architecture

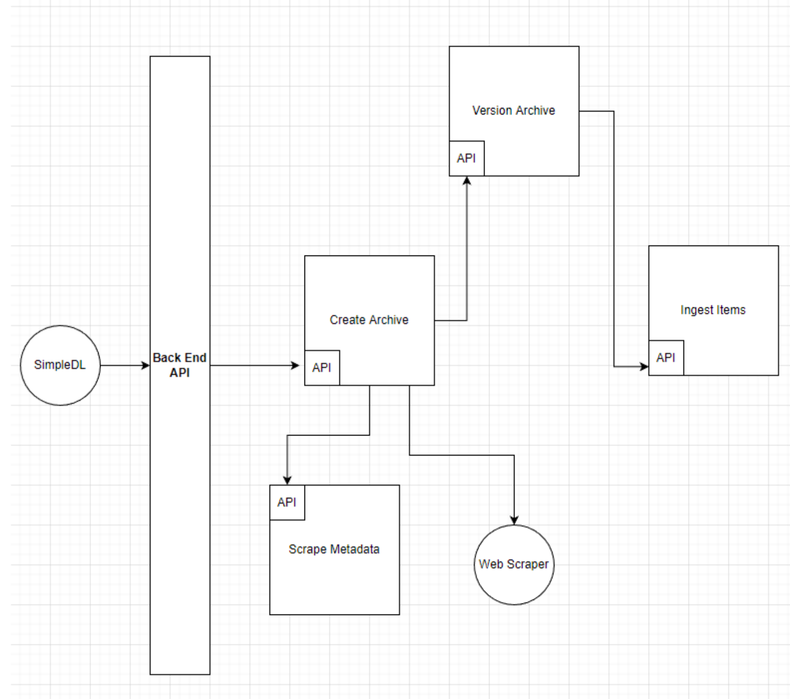


**Figure 2: Architecture Diagram depicting the service orientated design of the Ingesting tool. Note entities represented by circles are not part of the ingestion tool they just use it**

Figure 2 showcases the architecture used for this program. This architecture was chosen because the program can be divided easily be sub-divided into a series of interdependent tasks that each will contribute to the program output that will satisfy the functional requirements detailed earlier. These tasks can hence be developed as services and they will include: (1) Scraping Metadata; (2) creating an archive: which involves checking if said archive is already ingested and generating metadata for this archive which can be understood by Simpledl; (3) Versioning the ingested archive: which involves checking how many versions of this archive exist in SimpleDL and hence storing this version correctly and generating the appropriate metadata for this version; (4) Object ingestion: this service entails ingesting the actual scraped archive into Simpledl with all its digital objects and producing metadata for those objects. Each task is reliant on the output of the next adjacent task and they communicate to each other via simple API's. Figure 3, in the appendix, shows how each service interacts in practice.

This design approach has inherent loose coupling and high cohesion allowing the program to satisfy the non-functional requirements of understandability/maintainability which is important since the *Archive of Archives* is an experimental project meaning that others may work on it in the future.

# 4 System Development and Implementation

## 4.1 Software Development Methodology

For this project the agile methodology chosen, is a simplified version of feature driven development, since this was a one man project[11]. The reason this methodology is a good fit for this project is because it is based off a service orientated architecture. Each service will be developed as a small program that will take about 2 weeks to complete. This means that each major service can be seen as a feature hence making it easy to setup a feature backlog and divide that backlog into iterations. At the end of each iteration the service developed during that can then be integrated with the previously developed service.

Feature driven development methodology uses 5 steps[11]: (1) Develop an overall object model; (2) Build a features list; (3) Plan by feature; (4) Design by feature; (5) Build by feature. This project was done over a total of 4 iterations, with steps 1-3 being done in iteration 0 and steps 4-5 being repeated for each feature over 3 iterations

### 4.1.2 Iterations

*Iteration 0.* The work for this iteration has been mostly covered by the above requirements and analysis and design section(steps 1 - 2*).* The features detailed in there were divided up into 3 iterations.

*Iteration One(1 weeks):* This involved the development of the service that adds a new archive entity to the SimpleDL repository and generating metadata for that archive entity. This involved writing a simple API class that will be used by the UI to call this service. An error contingency, that reverts the SimpleDL repository to its' state before the ingestion of the new scraped archive in case an error occurs during ingestion, will also be developed.

*Iteration Two(1 week):* This involved the development of versioning service. This entails checking up to what version the scraped archive being ingested has been stored up to and using that information to store it correctly

*Iteration Three(1 week):* This iteration involved creating the Metadata Scraper, which scrapes just the metadata for the archive that is to be ingested into the *Archive of Archives* using the OAI-PMH protocol This is to be stored in a directory the Ingestor can access.

*Iteration Four(2 weeks.)* This involved the development of the service that ingests digital objects from a scraped archive into SimpleDL using the metadata generated from the metadata scraper. This entails writing a class that is able to understand the format of the scraped archive and the output of the Metadata Scraper, so that it can process each digital object and generate metadata that SimpleDL can understand for each object. An algorithm will also be developed in this iteration which will compare the contents in the scraped archive currently being ingested to the latest version of that archive in SimpleDL to determine: (1) how many objects were added or removed and (2) which objects were modified. This information will be added to the metadata for this version.

## 4.2 Language choice

The language chosen to develop the system was python. This language was chosen because: (1) it is a language that is familiar to the developer, hence simplifying the development process; (2) it is a language that is very easy to understand, making it easier for future developers of the *Archive of Archives* lives' easier; (3) Python has very robust XML parsing libraries.

## 4.3 System Implementation

In order to understand this implementation one needs to understand how SimpleDL's repository works. SimpleDL uses a flat file format repository instead of a database management system, meaning that instead of tables, directories are used. This ingestion system works with 3 key directories:

The *new directory*: this directory stores the output of the Archive Collector and hence is where the scraped archive is ingested from. This output of the Archive Collector was formatted as follows: A directory that contains 3 other directories. These directories were: (1) A directory that stored all the HTML pages for the archive; (2) A directory that stored a tree of directories based off of the structure of the scraped archive with all the digital objects and (3) a metadata directory(produced from the metadata scraper) that contained all the metadata for all the digital objects stored in (2).

The *spreadsheet directory*: this is the directory where the metadata for collections and digital objects are stored in a hierarchical fashion, with parent collections' metadata being stored in the top level directory and the individual items' metadata being stored in the lowest directory. The format in which this metadata is stored is CSV spreadsheets. For this project, a 3 level hierarchy of directories was used(as seen in Figure 4). The top level contains directories that store all data related to the archive they represent. The second level contains all versions of the top level archive; each version is stored as a directory in the second level. The third level contains just the csv file that stores the metadata for each individual object in that version

The *collection directory*: this is the directory in which the actual digital objects are stored in the same hierarchical fashion as in the *spreadsheet directory*.

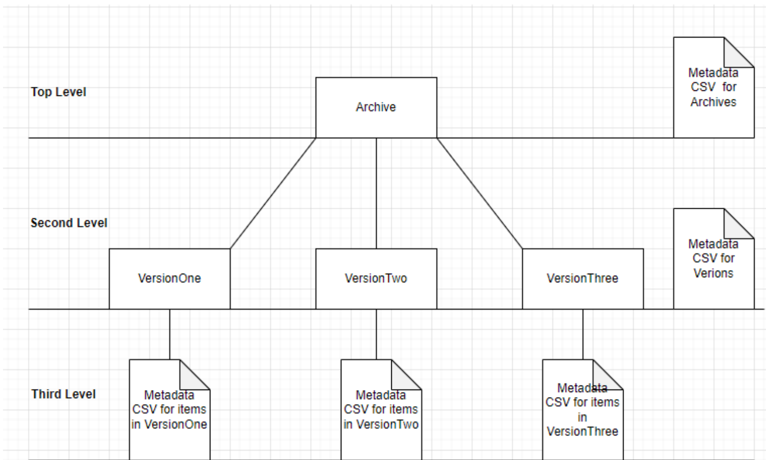


**Figure 4: Hierarchy Diagram showing the end result of the ingestion of three versions of one archive in the spreadsheets directory**

All classes have a similar structure- they each have a constructor and one public method that in turn invokes all the private methods to execute the service the class implements. Figure 4 shows a class diagram showing each class and their interactions, all methods referred to in the upcoming sections will be present Figure 5.

### 4.3.1 CreateArchive Class

This class represents the service that creates the archive directory in SimpleDL's flat file format repository and generates the appropriate metadata for this archive. How the class achieves this is by first scanning the *spreadsheet directory* to see if this archive has been ingested previously using the *checkArchiveExists()* Function. If it has not, then a new directory is made to house the versions and digital objects for this archive using the *createArchDirs()* function. The spreadsheet containing the metadata for each archive ingested will be updated with an entry containing the metadata for this archive using the *writeArchToSpreadsheet()* function. This function gets the required metadata by using the function *generateArchiveInfo().* This function is responsible for generating the metadata, for each metadata field, which will be written into the spreadsheet for this archive. Notable fields include *legacyId,* which is needed if this collection is a parent collection, *date* which is acquired using the datetime library and *description* which is information describing the archive provided by the user. The function *cleanArchive()* is invoked if a problem happens when adding the archive to revert the repository to the state it was in before the attempted ingestion of the scraped archive. This is achieved by having a try except block in the *addArchive()* function with the method *cleanArchive()* being in the except block. The *cleanArchive()* function removes the metadata entry in the spreadsheet for this archive as well as removing the directory created in the *spreadsheet directory* and the *collections directory* for this archive, if this archive was never ingested before or removes the version directory(and all of its' contents) for this version of the archive and removes the entry in the version spreadsheet, if this archive has already been ingested.

### 4.3.2 CreateVersion Class

This class implements a service that versions an archive being ingested by creating a directory in the second level of the hierarchy, in the *spreadsheets* and *collections* directory, which represents a version of the archive as well as all metadata for this version. Each version needs a unique *legacyId* since it is meant to be a parent collection. This is implemented in the versioning system by naming each directory in the version directory after their *legacyId.* The id generation is done by simply incrementing by 1 from a start number of 1 for each new version of the archive created. So, for example, if 2 versions of an archive exist, when a third version is created for that archive, it will be saved under the directory called 3. The *checkVersion()* function is used to calculate this id and the *createVersionDirs()* function is used to create the version directories in the second level of the hierarchy in the *collection* and *spreadsheet* directories. The *writeVersionToSpreadsheet()* function and the *generateVersionInfo()* do the same thing as the *writeArchToSpreadsheet()* function and the *generateArchiveInfo()* function in the *createArchive* class except with different metadata headings specific for a version. Notable metadata fields for each version include *numItemDifference* and *numItemsModified*. These fields document the change in item count and the number of items changed from the previous version, respectively. The *calculateItemDifference()* is used to find these values by using output received from the *ingestItems* class as well as searching the previous version's entry in the spreadsheet for the number of items it contained and subtracting the two values. The *moveZipClean()* function is responsible for moving the scraped archive from the *new* directory to its correct location in the *Collection* directory(Under the correct archive directory and in the correct version directory for that archive) in SimpleDL, compressing it in that directory aswell as removing the metadata directory created by the metadata scraper. In order to achieve this the *moveZipClean()* function uses: the *shutil.archive* function in the shutil library for compression; the *shutil.copy* function to copy the scraped archive to its correct location in SimpleDL and the *shutil.rmtree* function to delete the now ingested scraped archive in the *new* directory.

### 4.3.3 IngestItems Class

The purpose of this class is to provide the service of ingesting the items from the scraped archive into Simpledl. The *createItemSpreadSheet()* is the main function responsible for this. This function processes a directory of metadata items generated for every item in the scraped archive(this metadata is produced by the *metadataScraper class*). Each item's metadata is first

formatted into a standard format using the *formatXml()* function then, it is put through an XML parser and all relevant elements are extracted according to the predefind metadata headings that SimpleDL understands. Once all the required elements have been extracted for an item, they are written as an entry in the item csv file.
Since the ingestion process requires the processing of a large amount of items(depending on the archive), the above process was multithreaded. The multiprocessing library was used to achieve this. The directory of metadata items that need to be processed was divided up into smaller segments and each segment was given to a thread to process. The segment size is determined by taking the whole directory and dividing it by 16. Since each process item needs to be written to the same spreadsheet, race conditions can occur. To circumvent that a shared queue, which is a thread safe data structure was used. A thread that runs the *writeToSharedCSV()* function was also created to constantly check the queue and pop off any item entries that have been added. These item entries are then written to the CSV file. The queue used for this was called *EntryBuffer.*

One notable algorithm implemented in this class is the algorithm that determines: Which items have been modified since the previous version and how many. This algorithm, makes use of the sha256 hash function to hash the digital objects contents and add that hash as part of an item entry in the item csv file. The *hashObject()* function is responsible for this. This hash is then compared with the hash in the previous version for that particular item. This is achieved through the use of two hashtables. When the ingestItem class is created, the hash and identifier fields of the previous version's spreadsheet are loaded and inserted into two hashtables. The *createHashTableHash()* function, which creates a hash table that stores the hash of an item in the previous version of the archive being ingested, and *createHashTableId(),* which is a hashtable that stores the identifier of an item stored in the previous version of the current archive being ingested, are used to achieve this. Both of these functions make use of the *hashTableFun()* function, which is the function responsible for mapping an item to its position in the hash table. This function does this by simply converting each character in the item to its Unicode value, adding them together and modding it by the table size(500). Collisions in the hastable are handed by using chaining. The *searchHashTableId()* is then used by the threads to search the Id hash table to verify that the item being processed does exist in the previous version. The *searchHashTableHash()* is then used to find the hash for the corresponding item in the previous version and then compare it with the currently ingested item's hash. If the hashes differ then the current item is tagged as modified and the number of items modified counter is incremented.
The next feature this class implements is the ingestion report. This feature uses the *createItemSpreadsheet(),* the *writeReport()* and the *itemExists()* function. The *itemExists()* function checks that the item that the metadata represents exists in the scraped archive. The *createItemSpreadsheet()* function uses this function to insert items being ingested into a list of items that don't exist in the scraped archive. This list is then used by the *writeReport()* function to create a report that lists the identifiers for all the items that did not exist in the scraped archive. Once again, since this involved writing to a shared file, the *writeReport()* function was implemented as a thread, that like the *writeToSharedCSV()* function, constantly checks a queue to see if an item is present and then pops that item and writes it to a file. The queue used for this was called nonIngestBuffer.

The function *generateWebsiteInfo()*, is used to generate metadata for the directory that stores all the HTML pages in the scraped archive. Notable metadata fields include *digitalObjectPath,* which holds the filepath to location of the homepage of the website once ingested into SimpleDL.

#### 4.3.4 MetadataScraper Class

Since the Archive Collector does not scrape metadata for digital objects in an archive, this class needed to implement a service that scrapes just the metadata from the archive being scraped, by the Archive Collector and store it in the same file location as the scraped archive. To do this, the class makes uses of the OAI-PMH interface. OAI-PMH is a protocol that most archives support[12] that allows one to extract an archive's metadata for all of its' items in a specified metadata format. The *harvest()* function is responsible for the use of the OAI protocol. It does this by creating and sending an HTTP request to the archive's OAI, interface requesting the metadata, and for it to be formatted in the Dublin Core format. Initially when the metadata is scraped using the OAI-PMH it comes in batches, so the *createMetadataItems()* function processes each batch and splits them into individual metadata items and saves each item to its own file.

Each metadata item has an element called *identifier* which is a handle that points to the location of where that digital object is stored in the archive. When developing the Metadata Scraper, it was found that in some metadata items, from Dspace based archives, the *identifier* element was not present or did not provide the full handle. The next few methods: *changeIdentifiersDspace(), changeIdentifiersEprints(), findResource() and createXMLTag()* are all part of an algorithm that ensures that the *identifier* element present and complete. The *changeIdentifiersDspace(), findResource() and createXMLElement()* were used to solve this. The *changeIdentifiersDspace()* function was used to check if that identifier was present; if it wasn't then the element would have to be built. From examining how Dspace constructs its *identifier* elements, it was determined that the general form was "bitstream/handle/x/y/z". This means that the values x y and z need to be determined. Both x and y could easily be determined since they were present in the other *identifier* elements for this item. The *findResource()* function was used to find the value for z by following the path bitstream/handle/x/y (with the determined x and y values) to the directory in the scraped archive and extracting the name of the item that was present there (the z value). The *createXMLElement()* function is then used to create the new identifier element for that item's metadata. Once a valid *identifier* element is present the *changeIdentifiersDspace()* will format that

handle so that it now points to that item's location in the scraped archive in SimpleDL. The process described above was multithreaded in a similar fashion to what was done in the *ingestItems* class. The *setUpThreads()* function was used to achieve this. This function divided up the metadata items into segments and gave each segment to a thread. These threads once again had to write to a shared directory so the same queue method explained earlier was used with *saveXml()* being the function that access and writes data from that queue to the output metadata directory

Eprints based archives metadata item's was found, during the development of the metadata scraper, to not have the same *identifier* element problem the Dspace based archives metadata item's *identifier* elements had. The *changeIdentifiersEprints()* simply modifies the *identifier* element so it points to the location of the item in the scraped archive stored in SimpleDL.

#### 4.3.5 Key Libraries used

*OS library*. Its functions to do with directory processing and file path creation were used.

*Shutil library*. Its functions to do with advanced directory processing such as entire directory removal, moving directories and zipping directories.

*Padas Library*. Its functions to do with advanced CSV reading and writing were used.

*Hashlib Library*. Its sha256 hashing function was used.

*Lxml's Etree Library*. Its XML parser was used.

*Requests Library*. This was used to make HTTP requests to the OAI interface to scrape an items metadata.

*Multiprocessing*. This library was used to multithread the ingestion of items and the formatting of identifiers in the metadata scraper.

## 5 System Evaluation and Testing

Since this system forms part of the backend of the archive of archives the most appropriate evaluation methods are automated testing as opposed to user testing. This was done through the creation of a script that was be configured to run the program and compare the actual output of the program with the expected output over a number of test cases. The output that was tested was: (1) Have all the items been ingested from the scraped archive into SimpleDL?, (2) Have the correct directories been created in the correct places to correctly reflect versioning?, (3) Does the error report correctly reflect which items were not ingested?
The metadata scraper part of the Ingestor does not require any direct testing since it makes use of an already established protocol(OAI). In addition tests done on the Ingestor itself also evaluate the handle formatting part of the metadata scraper.

In addition to the functional testing above the non-functional requirements of: performance, scalability and portability will be evaluated. All tests were performed on a Windows based machine unless otherwise stated

### 5.1 Functional Test One: Archive Ingestion

#### 5.1.1 Test Case Description

This test will verify if the system can take a scraped archive produced from the archive collector and ingest it into the directory hierarchy in the SimpleDL repository and gernerate the appropriate metadata

#### 5.1.2 Test Case Data

- Scraped Archive called archive X, size 133 items generated by Eprints
- Scraped archive called archive Y, size 1115 generated by Dspace

#### 5.1.3 Test Result

| Test Case | Expected | Result |
|---|---|---|
| Ingest Archive X | Directories created in correct locations, appropriate entries added to CSV files, scraped archive moved correctly and 133 items ingested | Pass, see supplementary information, table one for output of the script |
| Ingest Archive Y | Same as above, except with 1115 items ingested | Pass, supplementary information, table one for output of the script |
| Ingest Archive X again | No new directory created in top level. New version directory created in second level for archive X. Version CSV updated | Pass, supplementary information, table one for output of the script |
| Ingest Archive Y Again | Same as above except for archive Y | Pass, supplementary information, table one for output of the script |
| Ingest Archive X 3 more times | No new top level directories. 3 new version directories created in 2nd level for archive X called 3, 4 ,5 | Pass, supplementary information, table one for output of the script |
| Ingest Archive Y 3 more times | Same as above but for archive Y | Pass, supplementary information, table |

| | | |
|---|---|---|
| | | one for output of the script |
| Change format of archive Y and ingest it | Exception to occur and repository to be reset | Pass, supplementary information, table one for output of the script |

## 5.2 Test Two(Functional) : Advanced Versioning

### 5.2.1 Test Case Description

This test case will focus on the versioning service. It aims to test whether the versioning can successfully detect how many items have been added and how many items have been removed from a previous version of the same archive

### 5.2.2 Test Case Data

- Scraped Archive called X

5.2.3 Test Cases

| Test Case | Expected | Result |
|---|---|---|
| Ingest Archive X twice | No items added/removed/modified, correct version and archive dirs created | Pass, see appendix for screenshots |
| Ingest Archive again X but remove 4 items | 4 items removed, 0 items modified, correct version and archive dirs created/present | Pass, see appendix for screenshots |
| Ingest Archive X(with -4 items) but add 4 items | 4 items added, 0 items modified, correct version and archive dirs created/present | Pass, see appendix for screenshots |
| Ingest Archive X but modify 3 items and remove 4 items | 0 items added/removed, 2 items modified, correct archive version dirs created/present | Pass, see appendix for screenshots |

## 5.3 Test Three: System and Performance Test

### 5.3.1 Test Case Description

This test will test the scalability as well as performance for the Ingestor and Metadata Scraper. Performance will be evaluated by timing the execution of the Metadata Scraper and Ingestor separately. Scalability can only be assessed through the size of archive that can be ingested, which will be a scraped archive with its number of items increased artificially by duplicating the metadata directory. Doing this will have the same affect as ingesting a scraped archive that actually has many items since the ingestor treats each item(whether its duplicate or not) as a new item and hence will overwrite the entry in item csv file if the processed metadata item is a duplicate.

### 5.3.2 Test Data

- **Scraped archive called archive X, size 132 itetms**
- **Scraped archive called archive Y, size 1115 items**

### 5.3.3 Test Cases

| **Test Case** | **Execution Time(Average over 3 runs in seconds)** |
|---|---|
| **Ingest Archive X** | **Ingestor: 5.67**<br>**Metadata Scraper: 9.73** |
| **Ingest Archive Y** | **Ingestor: 7.71**<br>**Metadata Scraper: 32.42** |
| **Ingest Archive X again** | **Ingestor: 5.75**<br>**Metadata Scraper: 9.41** |
| **Ingest Archive Y again** | **Ingestor: 7.22**<br>**Metadata Scraper: 32.48** |
| **Duplicate Items in Archive Y till 35681 then ingest one version** | **Ingestor: 51.84** |
| **Take above archive and ingest it again** | **Ingestor: 87.18** |

## 5.5 Test Four: SimpleDL integration

### 5.5.1 Test Description

This tests primary goal is to evaluate whether or not SimpleDL understands the metadata produced in each of the csv files . It will involve ingestion two versions of two scraped archives and seeing the hierarchy of folders SimpleDL creates when processing that metadata is the same as the hierarchy in the spreadsheets directory, with all digital objects present

### 5.5.2 Test Data

- Scraped Archive X, 133 items
- Scraped Archive Y, 1115 items

| **Expected Result** | **Actual Result** |
|---|---|
| Hierarchy Correct for archive X | Correct hierarchy |
| hierarchy Correct for archive Y | Correct hierarchy |

| All items present for version 1 archive X | All items present |
|---|---|
| All items present for version 1 of archive Y | All items present |

### 5.6 Discussion

From test one it can be seen that the Ingestor is able to ingest and version archives produced by the archive collector correctly. While only two archives were used to test this the results are still fairly reliable since all Dpsace and Eprints archive's have a similar format. Only 5 versions of each archive were ingested, this however is sufficient to show that the ingestor can ingest n versions of an archive long as version (n – 1) exists. The only pitfall that can result from this, is if version (n – 1) of an archive gets removed somehow. If another version of that archive is then ingested it will be saved under version (n – 1) instead of n. To solve this a simple check can be done before the ingestion of another version of an archive, to see if the latest version is in fact present.

From test two it can be seen that the versioning features: Number of items modified and Number of items added or removed works correctly. These results make sense since the number of items modified feature uses a precise hashing algorithm(sha256) to detect changes and to detect changes in the number of items between versions a simple subtraction is used.

From test three the efficiency/performance the ingestor component performs quite well since it makes use of multithreading. Using multithreading is important since it allows the Ingestor to be scaled which is important since the *Archive of Archives* is going to be used to rapidly store archives. It can be seen in test case 6 that ingesting an archive of size 35681 takes 87.18, if this was done sequentially this would take longer. However in order to fully leverage multithreading, a machine that has many cpu cores needs to be used. This may not always be possible for low resource environments, which could easily be a candidate for the use of the archives of archives since its built off SimpleDL. The functionality that requires the most computational power is what was tested in test two. While this functionality does provide an eloquent way of showing how an archive has changed over its' different versions it may not be strictly needed. Not having this functionality significantly decreases the computation time as seen by the difference in execution time from test 5 to test case 6. An option to hence turn this feature off should be added in the future.

The metadata scraper scrapes metadata of an archive using the OAI protocol. In test three it can be seen that the metadata scraper is acting as a performance bottleneck. This is because requests to the OAI interface are made sequentially, this however is difficult to parallelize since output from the previous OAI call(resumption token) is needed for the next one. This problem can however be circumvented when this part is integrated with the web scraper since the metadata scraper can be configured to run in parallel with the web scraper.

From test four it can be seen that the import script in SimpleDL is able to process an organize the metadata into the csv files to a hierarchy of metadata that SimpleDL will use for indexing. The main limitation with this test is the lack of integration with the UI component. Because of this tests that involve searchability of an item cant be performed.

One other notable limitation may be scalability with the current system. Test case three showed that the system is able to ingest a very large archive in a feasible amount of time however the system can only ingest one archive's items at a time. If one was planning to scale the archive of archives up to mass ingest archives the current ingestion and meta scraper would become a bottleneck. This problem can be solved by once again using multithreading however each thread will run the ingestor program for a different scaped archive.

## 6 Conclusions

This report showcased the development cycle for the ingestor and metascraper part of the archive of archives. From the results of the testing it is clear that the Ingestor is able to Ingest archives, in a feasible amount of time, in such a way that they are: (1) Versioned correctly, (2) Are not missing any items, (3) Correctly detect changes in items from previous versions and (4) understood by SimpleDL.

## 7 Future work

The Ingestor tool can be scaled up into a tool that is able to ingest multiple scraped archives at the same in a feasible amount of time.
The Metadata Scraper can be improved to scrape metadata from archives that do not support the OAI-PMH protocol
The Ingestor can have its algorithm that detects items modified improved to detect exactly what and where these changes occurred.
Complete integration with the UI and Archive Collector

## 8 References

1. Dimitrios A. Koutsomitropoulos, Anastasia A. Tsakou, Dimitris K. Tsolis, Theodore S. Papatheodorou.2004, Towards the Development of a General-Purpose Digital Repository, Vol: ICEIS (5)

2. Hussein Suleman. 2021, Simple DL: A toolkit to create simple digital libraries, International Conference on Asian Digital Libraries, 2021, Springer, 325-333. https://pubs.cs.uct.ac.za/id/eprint/1512/1/paper_88.pdf [8] Kumar, A., Saigal, R., Chavez, R. and Schwertner, N
3. Robert Tansley, Mick Bass, David Stuve, Margret Branschofsky, Daniel Chudnov, Greg McClellan, and MacKenzie Smith. 2003. The DSpace institutional digital repository system: current functionality. In Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries (JCDL '03). IEEE Computer Society, USA, 87–97.
4. Kurtz, M. (2010). Dublin Core, DSpace, and a brief analysis of three university repositories. Information technology and libraries, 29,1, 40-46.
5. Khan, S. 2019. DSpace or Fedora: Which is a better solution?. SRELS Journal of information management, 56,1, 45-50
6. Staples, T., Wayland, R. and Payette, S., 2003. The Fedora Project. D-Lib Magazine, 9(4). DOI https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.919 4&rep=rep1&type=pdf
7. Kumar, A., Saigal, R., Chavez, R. and Schwertner, N., 2004. Architecting an Extensible Digital Repository. In ACM/IEE CS joint conference on digital libraries. JCDL. DOI https://dl-acmorg.ezproxy.uct.ac.za/doi/10.1145/996350.99635
8. Suleman, H. (2007). in-Browser Digital Library Services, in Kovacs, Laszlo, Norbert Fuhr and Carlo Meghini (eds): Proceedings of Research and Advanced Technology for Digital Libraries, 11th European Conference (ECDL 2007), 16-19 September, Budapest, Hungary, pp.462-465, Springer
9. Hussein Suleman (2019) Investigating the effectiveness of client-side search/browse without a network connection, Proceedings of 21st International Conference on Asia-Pacific Digital Libraries (ICADL), 4-7 November 2019, Kuala Lumpur, Malaysia, Springer. Available DOI
10. Ian H. Witten, Stefan J. Boddie, David Bainbridge, and Rodger J. McNab. 2000. Greenstone: a comprehensive opensource digital library software system. In Proceedings of the fifth ACM conference on Digital libraries (DL '00). Association for Computing Machinery, New York, NY, USA, 113–121. https://doi-org.ezproxy.uct.ac.za/10.1145/336597.336650
11. Goyal, S. (2008). Major seminar on feature driven development. *Jennifer Schiller Chair of Applied Software Engineering*, http://csis.pace.edu/~marchese/CS616/Agile/FDD/fdd.pdf
12. Lagoze, C., & Van de Sompel, H. (2003). The making of the open archives initiative protocol for metadata harvesting. *Library hi tech* .https://www-emerald-com.ezproxy.uct.ac.za/insight/content/doi/10.1108/07378830310479776/full/pdf?title=the-making-of-the-open-archives-initiative-protocol-for-metadata-harvesting
13. Khan, S. 2019. DSpace or Fedora: Which is a better solution?. SRELS Journal of information management, 56,1, 45-50.

## 9 Supplementary information

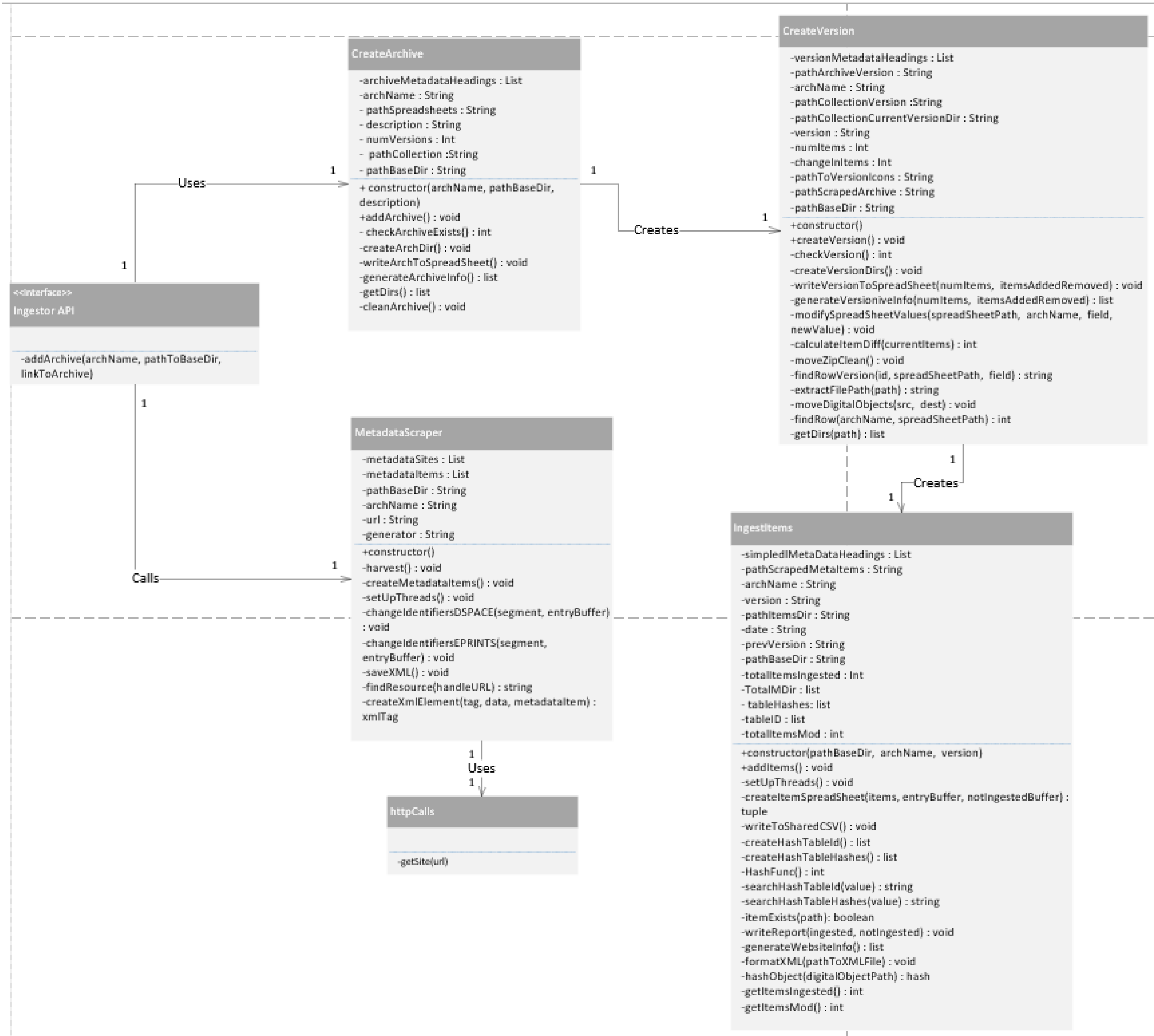


**Figure 5: Class Diagram showing the classes that make up the Ingestor and Metadata Scraper and their interactions.**

| Test Case | Screenshot |
|---|---|
| 1 | Directory for archive X has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, ]<br><br>Executing tests for create version service...<br>Version directory for version 1 has been correctly created in the second level in the spreadsheets and collection directory for archive X<br>Entry with legacyID 1 has been correctly added to the version CSV<br>Archive compressed and moved correctly to its location in the collection directory<br>Current directories present in the version directory for archive X [ 1, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 133 items ingested |
| 2 | Directory for archive Y has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId Y has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, Y, ]<br><br>Executing tests for create version service...<br>Version directory for version 1 has been correctly created in the second level in the spreadsheets and collection directory for archive Y<br>Entry with legacyID 1 has been correctly added to the version CSV<br>Archive compressed and moved correctly to its location in the collection directory<br>Current directories present in the version directory for archive Y [ 1, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 1115 items ingested |
| 3 | Directory for archive X has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, Y, ]<br><br>Executing tests for create version service...<br>Version directory for version 2 has been correctly created in the second level in the spreadsheets and collection directory for archive X<br>Entry with legacyID 2 has been correctly added to the version CSV<br>Archive compressed and moved correctly to its location in the collection directory<br>Current directories present in the version directory for archive X [ 1, 2, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 133 items ingested |
| 4 | Directory for archive Y has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId Y has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, Y, ]<br><br>Executing tests for create version service...<br>Version directory for version 2 has been correctly created in the second level in the spreadsheets and collection directory for archive Y<br>Entry with legacyID 2 has been correctly added to the version CSV<br>Archive compressed and moved correctly to its location in the collection directory<br>Current directories present in the version directory for archive Y [ 1, 2, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 1115 items ingested |
| 5 | Directory for archive X has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, Y, ]<br><br>Executing tests for create version service...<br>Version directory for version 5 has been correctly created in the second level in the spreadsheets and collection directory for archive X<br>Entry with legacyID 5 has been correctly added to the version CSV<br>Archive compressed and moved correctly to its location in the collection directory<br>Current directories present in the version directory for archive X [ 1, 2, 3, 4, 5, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 133 items ingested |
| 6 | Directory for archive Y has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId Y has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, Y, ]<br><br>Executing tests for create version service...<br>Version directory for version 5 has been correctly created in the second level in the spreadsheets and collection directory for archive Y<br>Entry with legacyID 5 has been correctly added to the version CSV<br>Archive compressed and moved correctly to its location in the collection directory<br>Current directories present in the version directory for archive Y [ 1, 2, 3, 4, 5, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 1115 items ingested |

**Table One: Output for test One**

<table>
<tr><th>Test Case</th><th>Screenshot</th></tr>
<tr><td>1</td><td>Executing tests for createArchive service....<br>Directory for archive X  has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, ]<br><br>Executing tests for create version service...<br>Total number of items modified since previous version: 0<br>Total number of items added/removed since previous version: 0<br>Current directories present in the version directory for archive X [ 1, 2, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 133 items ingested</td></tr>
<tr><td>2</td><td>Directory for archive X  has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, ]<br><br>Executing tests for create version service...<br>Total number of items modified since previous version: 0<br>Total number of items added/removed since previous version: -4<br>Current directories present in the version directory for archive X [ 1, 2, 3, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 129 items ingested</td></tr>
<tr><td>3</td><td>Executing tests for createArchive service....<br>Directory for archive X  has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, ]<br><br>Executing tests for create version service...<br>Total number of items modified since previous version: 0<br>Total number of items added/removed since previous version: 4<br>Current directories present in the version directory for archive X [ 1, 2, 3, 4, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 133 items ingested</td></tr>
<tr><td>4</td><td>Executing tests for createArchive service....<br>Directory for archive X  has been created in the top level directory in the spreadsheets and collections directories<br>Entry with legacyId X has been added to the archive CSV<br>Current directories present in the top level directory: [ archive.csv, X, ]<br><br>Executing tests for create version service...<br>Total number of items modified since previous version: 2<br>Total number of items added/removed since previous version: 0<br>Current directories present in the version directory for archive X [ 1, 2, 3, 4, 5, version.csv, ]<br><br>Executing tests for item ingestion...<br>Test Successful, a total of: 133 items ingested</td></tr>
<tr><td></td><td></td></tr>
</table>

**Table Two: Output for test 2**

| Test Case | Screenshot |
|---|---|
| 1 | Total Metadata scrape time: 10.240773916244507<br>Total time for archive Ingestion: 5.654999494552612<br>Total Metadata scrape time: 9.589404106140137<br>Total time for archive Ingestion: 5.718002080917358<br>Total Metadata scrape time: 9.374065399169922<br>Total time for archive Ingestion: 5.6459996700286865 |
| 2 | Total Metadata scrape time: 32.6250388622228394<br>Total time for archive Ingestion: 7.136007308959961<br>Total Metadata scrape time: 32.52698087692261<br>Total time for archive Ingestion: 7.039001703262329<br>7.020000219345093<br>Total Metadata scrape time: 32.09312701225281<br>Total time for archive Ingestion: 7.0470099449157715 |
| 3 | Total Metadata scrape time: 9.520316362380981<br>Total time for archive Ingestion: 5.777998447418213<br>Total Metadata scrape time: 9.787029027938843<br>Total time for archive Ingestion: 5.804001092910767<br>Total Metadata scrape time: 8.91424822807312<br>Total time for archive Ingestion: 5.659066200256348 |
| 4 | Total Metadata scrape time: 32.42146301269531<br>Total time for archive Ingestion: 7.173004150390625<br>Total Metadata scrape time: 32.5454216003418<br>Total time for archive Ingestion: 7.142000913619995<br>Total Metadata scrape time: 32.48256778717041<br>Total time for archive Ingestion: 7.338999509811401 |
| 5 | 50.76169776916504<br>Total time for archive Ingestion: 50.908695697784424<br>50.38791751861572<br>Total time for archive Ingestion: 50.50191950798035<br>54.3770911693573<br>Total time for archive Ingestion: 54.488091230392456 |
| 6 | 82.88199925422668<br>Total time for archive Ingestion: 85.77100348472595<br>84.89352750778198<br>Total time for archive Ingestion: 87.724529504776<br>85.23500204086304<br>Total time for archive Ingestion: 88.04600048065186 |

**Table 3: Output for test 3**

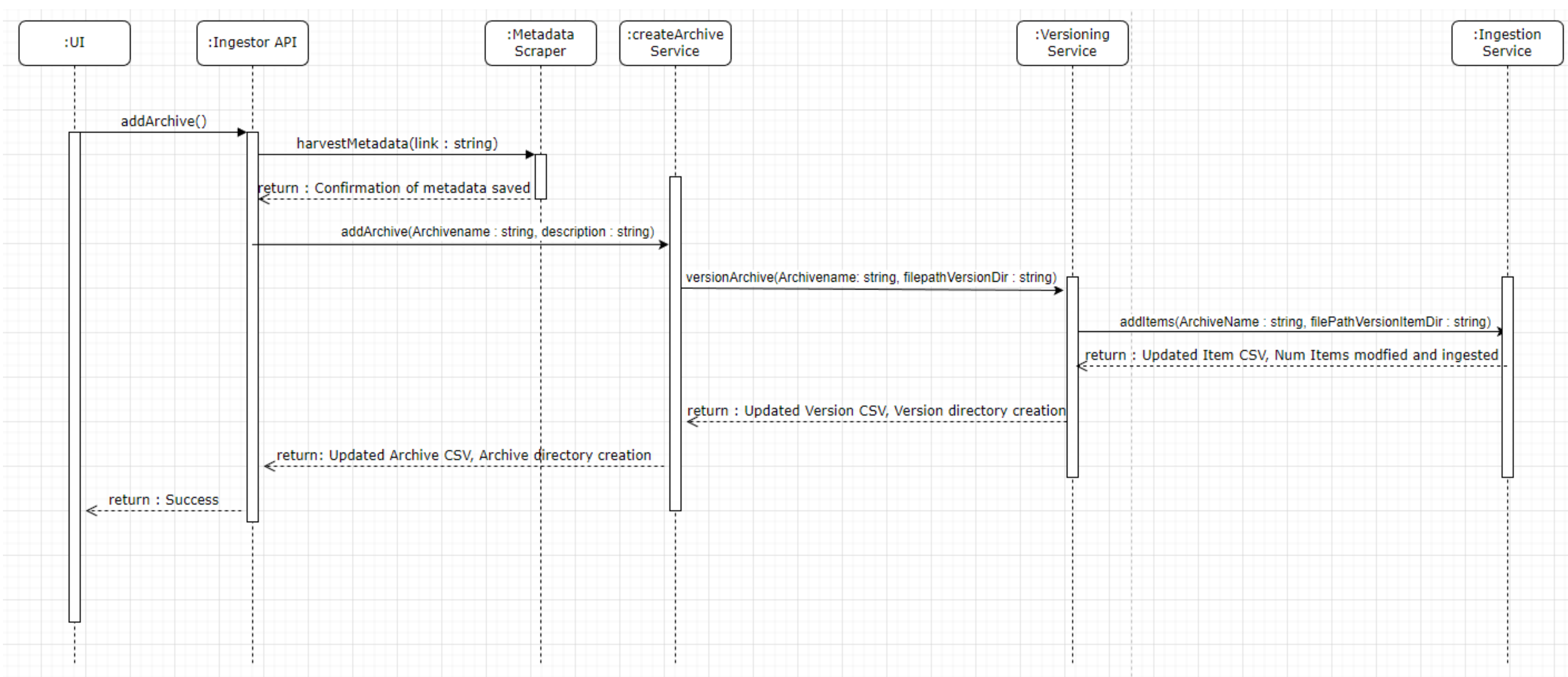


**Figure 3 Sequence Diagram depicting how and which services invoke each other**

**Figure 6: Hierarchy Output for Test 4**