
# Assignment 2: Modeling

## Software Design



| **Team member** | **Student #** | **Email** |
|---|---|---|
| Charlee Lachance | 2836195 | c.l.lachance@student.vu.nl |
| Luca Snoey | 2835683 | l.s.snoey@student.vu.nl |
| Anna Serbina | 2835638 | a.s.serbina@student.vu.nl |
| Jacob Roberts | 2837670 | j.r.roberts@student.vu.nl |



---

Note: in the following document, **Classes** will be written in bold, **Packages** in underlined bold, *ClassAttributes* and *ClassMethods()* in italics, ***States*** in bold italics, `messages` in monospace, and Relationships in underlined.

# 1 – Summary of Changes from Assignment 1

*Author: Charlee Lachance*

**Added more technical details:** In general, the feedback stated that we nicely overviewed our system but could have gone into a bit more detail. We added some examples of methods and classes that we plan to use during implementation. We also hope that the increased depth of Assignment 2 is enough to clarify anything that was necessarily vague in the early stages of Assignment 1.

**Updated information about implementation plans:** While working on Assignment 2, it became clear that some of our plans (e.g. for the encryption process) were misguided. After doing research into different libraries and methods available to us, we have removed password hashing from the encryption slide as this will no longer be necessary. We also made minor changes to other slides to include more up-to-date implementation plans or logic flows.

**Elaborated on the extraction process:** Tanav advised us to give more explanation on how we will implement extraction with passwords. We updated the Feature 2 slide to better explain how extraction will work for both encrypted and unencrypted archives.

**Assigned all team members to at least one feature:** Tanav requested for us to distribute the features evenly between the team members. While we plan to collaborate throughout the coding process, everyone will be focusing on at least one feature in particular.

# 2 – Class Diagram

*Author: Charlee Lachance*

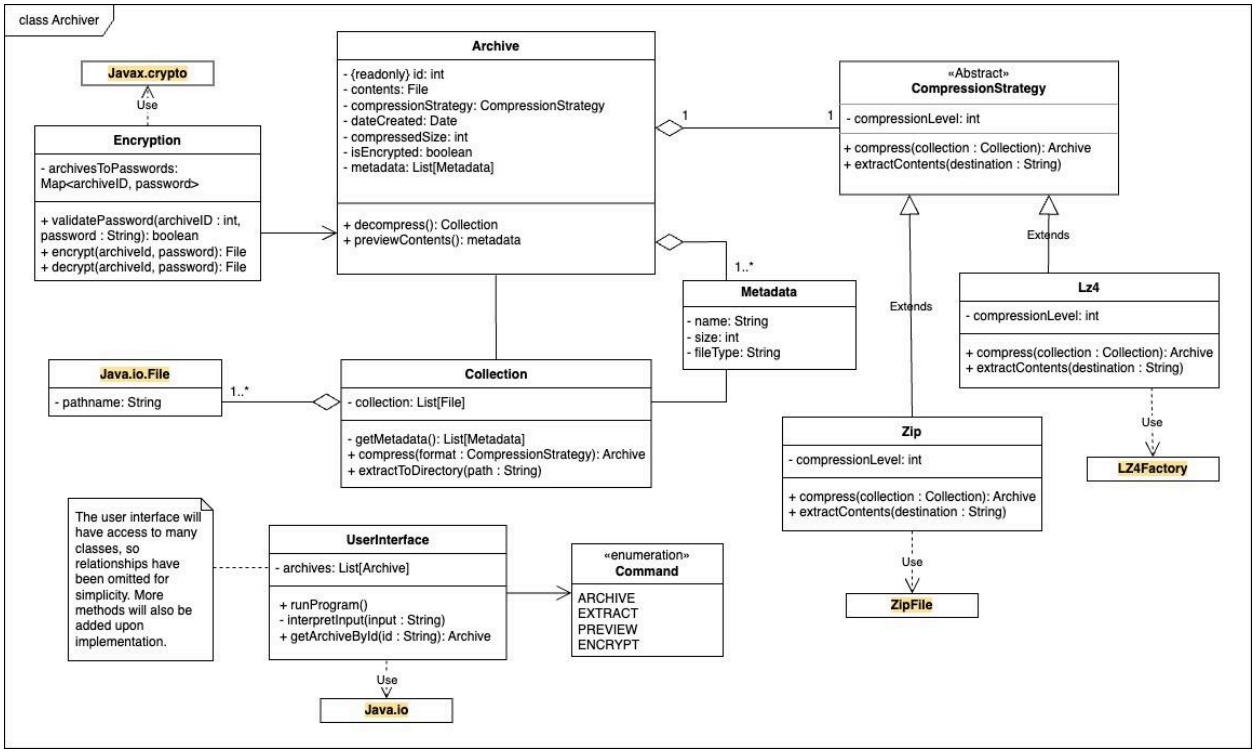


Figure 1: Class Diagram

**UserInterface:** This Singleton class will control the logic of interacting with the user. It will be responsible for printing instructions for the user (appropriate methods will become clear upon implementation) and interpreting the user's commands through *interpretInput()*. As there will only be one instance of this class during runtime, we have decided to store a list of active archives in the *archives* attribute. An alternative to this would be to maintain this list in a new class. However, since this class would only have one attribute and a couple of methods at most (*getArchiveById() and getArchives()*), it will reduce overall complexity to combine it into an existing Singleton class. Please note that as this class control's most of the program's logic, it will have access to many other classes in the diagram such as **Archive**, **Collection**, and **Encryption**. These relationships have been omitted from the diagram to increase clarity.

➔ **Command:** This enumeration will be used by **UserInterface** to help control the logic flow based on the user's input. Each element represents one of the acceptable commands a user may enter. **UserInterface** will compare the input to the items of **Command**, and will then take the relevant course of action, which may involve prompting the user to enter additional information.

- ➔ **Java.io: UserInterface** will use this Java package to help deal with user input. The desirable classes of this library will be determined during implementation.

**Collection:** This class serves as an intermediary between the user and the contents of an archive. Although we could have chosen to create an **Archive** object directly from **UserInterface**, we feel that this additional class will add organization to a potentially large number of files and folders. Furthermore, we will need to perform an analysis of the files, which would be much more difficult to do in a compressed form. Once a user has selected the files they would like to add to an archive, a **Collection** object will be created via **UserInterface**. The constructor will call *getMetadata()*, and this information will later be passed on to the new **Archive** object formed by *compress()*, hence the associations to both **Metadata** and **Archive**. Once the **Archive** is created, no further references to this **Collection** object are necessary. The second use case of **Collection** will be to briefly store files from an archive after decryption and decompression. Note that this creates a new object, bearing no relation to the **Collection** object that created the archive. This is to avoid retaining the files and wasting memory.

- ➔ **Java.io.File:** We will use Java's **File** class to represent the multitude of files when not archived, as well as to represent the compressed file. Because one or more instances of **File** compose a **Collection**, but can still exist independently, these two classes share an aggregation relationship.

**Metadata:** Instances of the **Metadata** class will store the necessary information for the "preview contents" feature. It stores the *name*, *size* (in bytes), and file *type*. As mentioned in the **Collection** description, this data will be collected before the files are compressed. Each file or folder will generate one **Metadata** object, and a list of these objects will be stored in **Archive**'s *metadata* attribute, giving rise to an aggregation relationship. When *previewContents()* is called, this list will be returned and printed to the user through **UserInterface**. Maintaining this information throughout the lifetime of an **Archive** object is a less complex alternative than somehow extracting this information from the compressed file upon request.

**Archive:** An instance of **Archive** will store all necessary information about the compressed file, including encryption status, size, and compression strategy. It will be created from a **Collection** object using a specific instance of **CompressionStrategy**. We have decided to make encryption of an archive optional, as not all users will have need for this layer of security. To promote information hiding, the only encryption information accessible to **Archive** is its own *isEncrypted* status – everything else will be contained in **Encryption**. The method *previewContents()* may be called by **UserInterface** if requested by the user, and *decompress()* for extraction of contents (**UserInterface** will decrypt the contents through **Encryption** first if *isEncrypted* is true).

**Encryption:** This class handles the optional functionality for encryption/decryption of archives. It keeps an internal structure *archivesToPasswords*; upon encryption of an archive, a user will be prompted to choose a password that will be used as a key in the encryption process. In order to extract the archive, the user will need to enter that same password and it will be checked against *archivesToPasswords* for correctness before the decryption process begins. **Encryption** has an association with **Archive** that is navigable in one direction. To increase modularity, **Encryption** will take care of the encryption/decryption process, and will update the *contents* and *isEncrypted* status of **Archive** when complete.

- ➔ **Javax.crypto: Encryption** will be using this library to assist with the encryption and decryption algorithms. More technical details of which classes we will be using will be determined during implementation.

**CompressionStrategy:** The **CompressionStrategy** class contains the information and methods related to compression and decompression for a single **Archive**, hence the 1:1 multiplicity relationship. We are storing an instance in **Archive** so we can refer back to it during extraction (as the extraction process is also dependent on compression format). We chose to make **CompressionStrategy** an abstract class because we wanted to ensure that future additions of new compression formats will require minimal changes to the codebase. Although each compression format obviously requires different implementations, they all share the same interface. Therefore, the technicalities relating to compression formats will not matter to the rest of the application. *CompressionLevel* is a configuration that the user will be prompted to choose.

- ➔ **Zip** and **Lz4:** These are the two compression formats that we will initially be supporting. They extend the **CompressionStrategy** abstract class to provide the relevant functionality.
- ➔ **Zip4J** and **LZ4Factory:** We will be importing **jz4-java** and **zip4j** from GitHub to simplify our compression code. **Zip4J** and **LZ4Factory** are the two most important classes that we will make use of.

# 3 – Package Diagram

*Author: Luca Snoey*

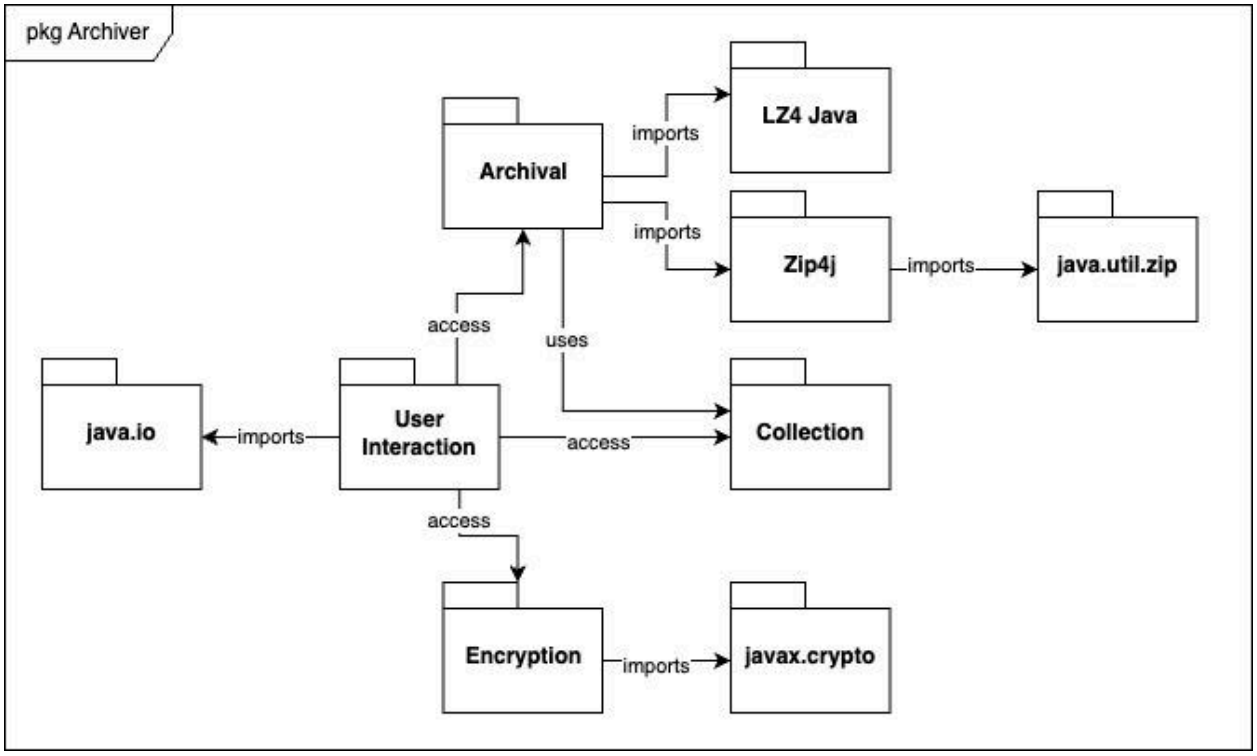


Figure 2: Package diagram

Here, the package diagram outlines our project structure and the external package dependencies we will rely on for our project functionality. The project structure is separated into 4 main bespoke packages (**Archival**, **User Interaction**, **Encryption**, **Collection**).

**Archival** covers all classes that work on compression and decompression of collection objects. **Archival** requires imports of LZ4 libraries and Zip4J libraries to facilitate archival processes in different formats. **Zip4J** was decided upon due to its more flexible and convenient usage for Java as a slight modification of **java.util.zip**, which it naturally imports. **Archival** uses **Collection** to package files for compression and stores previous archives in a list, as well as file metadata.

**User Interaction** represents the command line interaction and the handling of user inputs. Class functionality includes prompt management, input processing, displays, etc… **User Interaction** necessarily imports **java.io** for command line inputs and accesses the **Collection**, **Encryption**, and **Archival** packages to interact between the user and those functionalities. From the command line all other classes will be accessed and their methods ran.

**Encryption** represents the class that will handle the archive encryption and decryption capabilities and password creation and hashing in cooperation with the command line. The encryption package imports **javax.crypto** in order to complete encryption functionalities.

**Collection** represents the class and class functionality of combining files into a group to be compressed. **Collection** communicates with the command line to add and remove files from a group and also represents the functionality of creating a collection from the decryption process.

# 4 – Object Diagram

*Author: Anna Serbina*

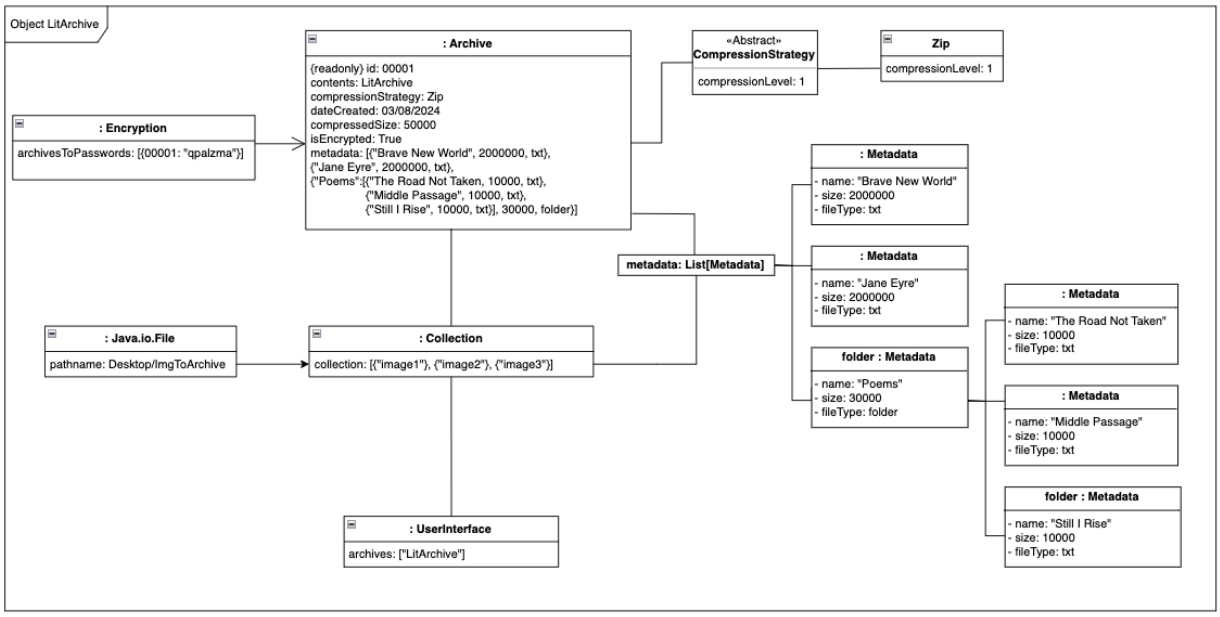


Figure 3: Object Diagram

The object diagram illustrated in Figure 3 provides a snapshot of our system, capturing instances of a LitArchive object and a collection of images to depict relationships within the system. The user initiated the archiving of a collection of files, resulting in the creation of a LitArchive – an instance of the **Archive** class. This archive, assigned a unique ID (00001), has a creation date of 03/08/2024. The archive's contents include the LitArchive (its name), and its compressed size is 50000000 bytes. The user selected the compression format as Zip, reflected in the compressionStrategy field. The archive also contains information about its contents in the list of metadata. In the case of compressing files and folders, metadata includes details such as file names, sizes, and types.

For example, the file "Brave New World" has an uncompressed size of 2000000 bytes and is of type txt. The folder "Poems" has an uncompressed size of 30000 bytes and contains a list of singular poems with associated metadata ("The Road Not Taken," "Middle Passage," "Still I Rise"). Since the archive is compressed, the **compressionStrategy** and its format (Zip) have an associated compressionLevel of 1.

After archive creation, its name is stored in the archives collection, an instance of the **UserInterface**: ["LitArchive"].

At a subsequent point, the user opted to encrypt the archive using the password "qpalzma." The LitArchive's isEncrypted variable was then set to True. The program updated the

archivesToPassword, an instance of the **Encryption** class, resulting in a map entry {00001: "qpalzma"}.

Upon the creation of the LitArchive archive, the associated collection was deleted. Our archiver ensures that **Collection** and **Archive** are mutually exclusive for the same collection of files. However, at the time of this system snapshot, a collection of images exists. This implies that the user uploaded them as a **Collection** but has not yet compressed them. Therefore, the existing pathname in the system is associated with the collection of images ("Desktop/ImgToArchive") and not the textfiles that are archived. No metadata is currently associated with the images; the user cannot call the getMetadata() method on the **Collection**, and the list of metadata is only created upon encryption.

# 5 – State Machine Diagrams

Our archiver application primarily revolves around archives, with most other classes emphasizing functionality rather than data storage. Consequently, the two key classes that undergo significant transitions and elucidate the most crucial states of the objects are the **Archive** and **Collection** classes. As a result, we chose them for state machines visualization. Despite their simplicity, these two diagrams effectively outline the majority of the inner workings of the archiver software.

## 5.1 Archive class

*Author: Anna Serbina*

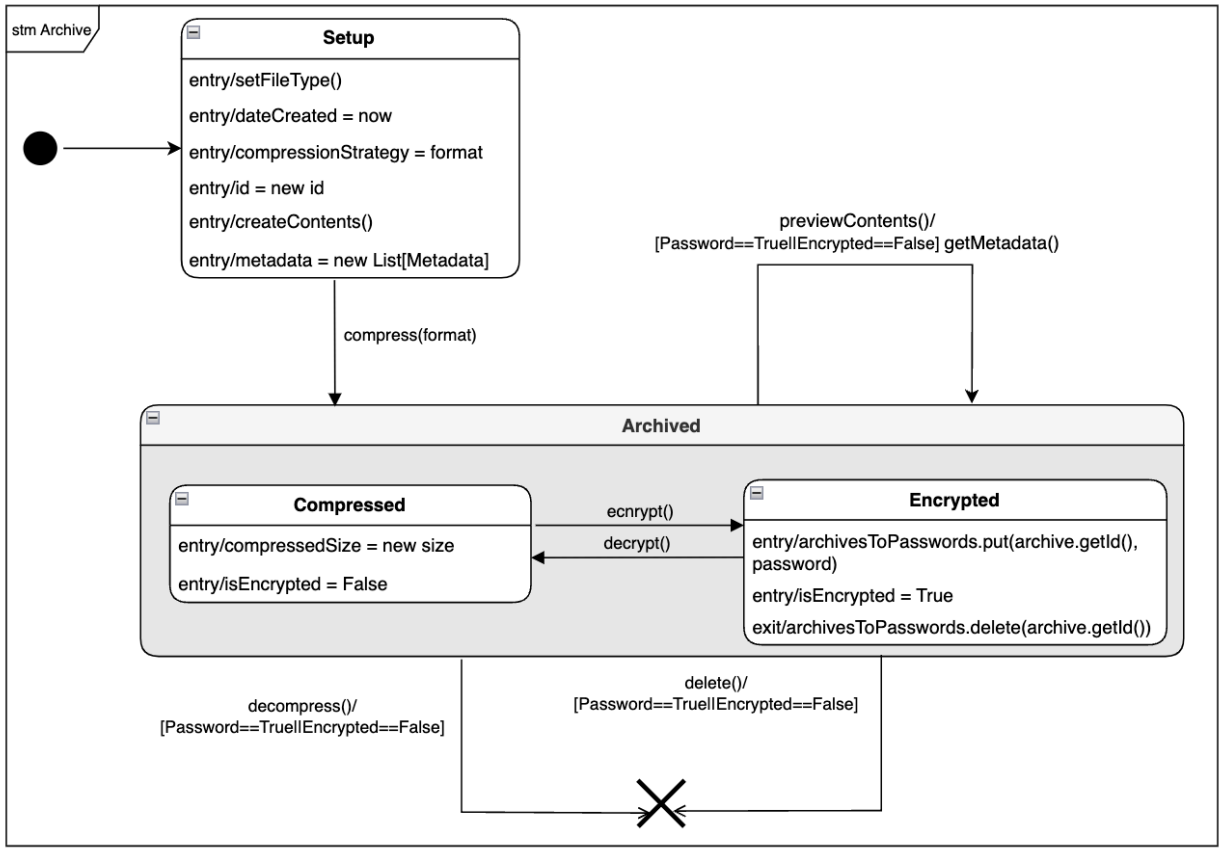


Figure 4: State Machine Diagram of the Archive Class

When an instance of the **Archive** class is first created, it enters the ***Setup*** state. In this phase, as the **Collection** triggers the creation of the archive, crucial information is collected. This information encompasses the name of the associated collection, the contents of the collection, the compression format specified by the user, and metadata from the collection to support the preview function.

To facilitate the *previewContents()* function when archiving the collection, we have chosen to duplicate metadata containing details such as the name, size, and file type of every file in the collection, along with its internal architecture, before compression. This enables users to preview the archive contents without the necessity of decompression and/or encryption. The *dateCreated* field is then set to the current time. Additionally, a *uniqueID* is generated for the archive in preparation for subsequent processing.

After the ***Setup*** state, the archive progresses through the archiving process based on the specified compression format, transitioning to the ***Compressed*** state. Upon entry into the ***Compressed*** state, the boolean *isEncrypted* is set to False, and the compressed size of the archive is recorded.

If the user chooses encryption, the instance transitions to the ***Encrypted*** state, prompting the user to establish a password for encryption. Upon entry into the ***Encrypted*** state, the *isEncrypted* boolean is set to True, signifying encryption. The user-provided password is then securely stored in the *archivesToPasswords* map. The user can encrypt the archive at any time using the *encrypt()* method.

Users retain the flexibility to switch from the ***Encrypted*** state back to the ***Compressed*** state using the *decrypt()* method. This method not only transitions the archive but also deletes the corresponding password from the *archivesToPasswords* map, ensuring a secure and reversible transition between encryption states.

Both the ***Compressed*** and ***Encrypted*** states fall under the category of the ***Archived*** composite state. The ***Archived*** composite state denotes a compressed folder of files, where the ***Compressed*** state is neither encrypted nor protected by a password, while the ***Encrypted*** state represents an encrypted, password-protected archive.

***Compressed*** and ***Encrypted*** states offer the user the following options:

1. *Decompress()*:
If the archive is encrypted, the user is prompted for a password. A correct password initiates the decompression of the archive into a collection object, resulting in the termination and deletion of the archive, and the extraction of files into a collection. An incorrect password leads to no action. If the archive is not encrypted, the archive is decompressed, and the files are extracted without any additional steps.

2. *previewContents():*
If the archive is encrypted, the user is asked to input a password for decryption. Once the correct password is provided, a preview of the archive is shown, featuring its internal architecture, file size, file type, and names. The *previewContents()* function makes use of the metadata that was

earlier copied from the associated **Collection**. If the archive is not encrypted, *previewContents()* functions seamlessly. In the event of an incorrect password, no action is taken.

3. *delete():*
If the archive is encrypted, the user is prompted to enter a password for decryption. Upon entering the correct password, the program initiates a secure process to delete both the archive and its internal files: all internal variables associated with the archive, such as references and metadata, are set to null, effectively erasing any trace of the archive within the program. In the case of an incorrect password, no action is taken to prevent unintended data deletion.

For non-encrypted archives, the deletion process is straightforward, and the archive contents are seamlessly removed from the program. This involves setting all internal variables to null, erasing all associated files and metadata without the need for encryption-related steps.

## 5.1 Collection class

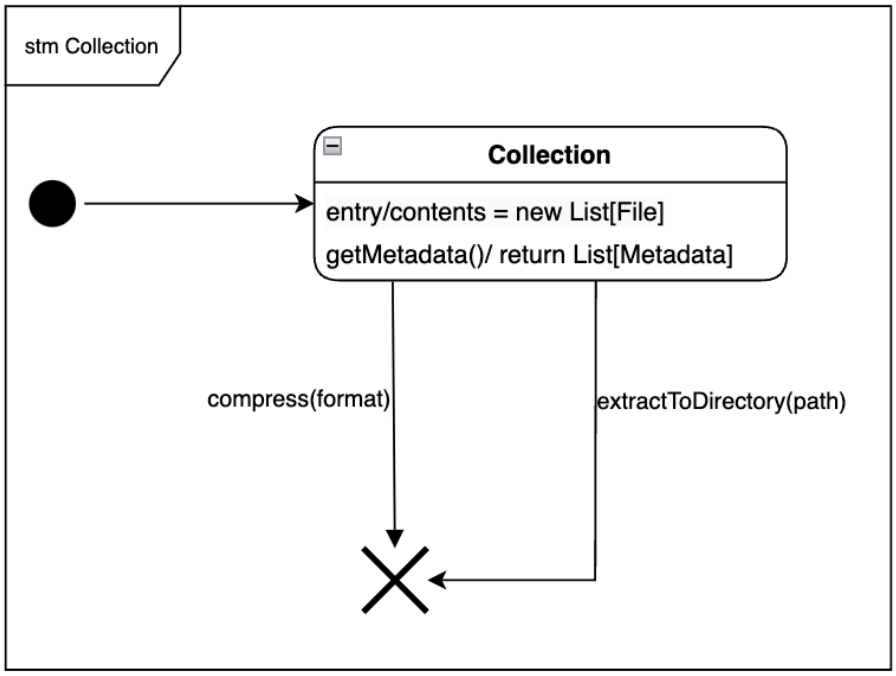


Figure 5: State Machine Diagram of the Collection Class

When a ***Collection*** object is instantiated, it retains its contents as a list of **Files**. Upon the *getMetadata()* method call, a metadata list is generated, representing a shallow copy of the internal structure of the collection, including the name, file type, and sizes of its files. This list plays a crucial role in the archival process, particularly in supporting the *previewContents()* function in the **Archive** class as described in the **Archive** state machine diagram.

When the *compress(format)* method is invoked on a ***Collection***, it undergoes compression based on the specified format, resulting in the creation of a new **Archive** object. The ***Collection*** object becomes obsolete and is terminated – the instance variable contents is set to null, effectively concluding the ***Collection*** state machine.

If the *extractToDirectory(path)* method is called on the ***Collection*** object, its contents are saved locally on the user's computer at the specified path. Subsequently, the contents instance variable is set to null, concluding the ***Collection*** state machine, and rendering the ***Collection*** object nonexistent.

# 5 – Sequence Diagrams

*Author: Jacob Roberts*

## 5.1 Creating an Archive

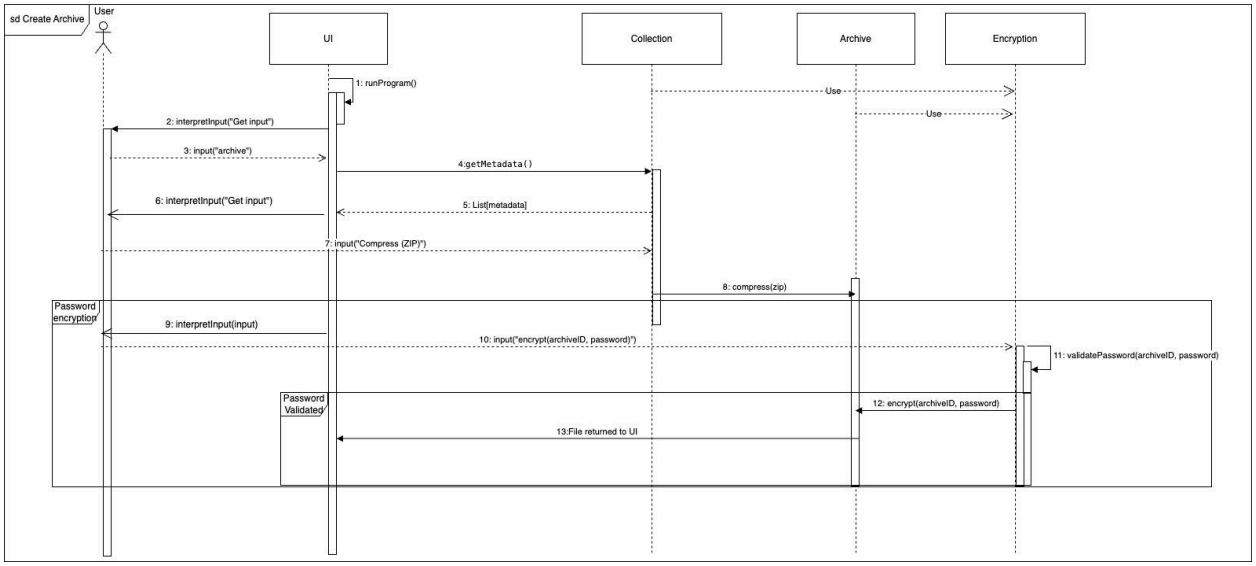


Figure 6: Sequence diagram of creating an archive

In this sequence, we are showing the process of creating a compressed **Archive** from a **Collection** of files with the option of **Encryption** using a password.. The sequence begins with the **User Interface**. It prompts the user to input commands and select files for archiving.

Once the user chooses to **Archive** files, the **User Interface** asks for the file paths and **Metadata** of these files. After returning a list of the **Metadata** to the **UI (UserInterface)**, the user then inputs a command *compress(ZIP)* with ZIP specifying the **CompressionStrategy**. This info is passed from the **UI** to the **Collection** class. From there, *compress(ZIP)* is called on the **Archive** class.

The next frame in the diagram shows what happens when the user wants to use **Encryption** and to password protect their archive that was just created and is the final compressed folder. The **UI** will interpret that the user wants to create a password and pass that string to the **Encryption** class. From there. **Encryption** will call *validatePassword(),* and when validated, will *encrypt()* the archive. In the end, the **UI** confirms the creation of the archive and returns the file to the user.

## 5.2 Extracting an archive

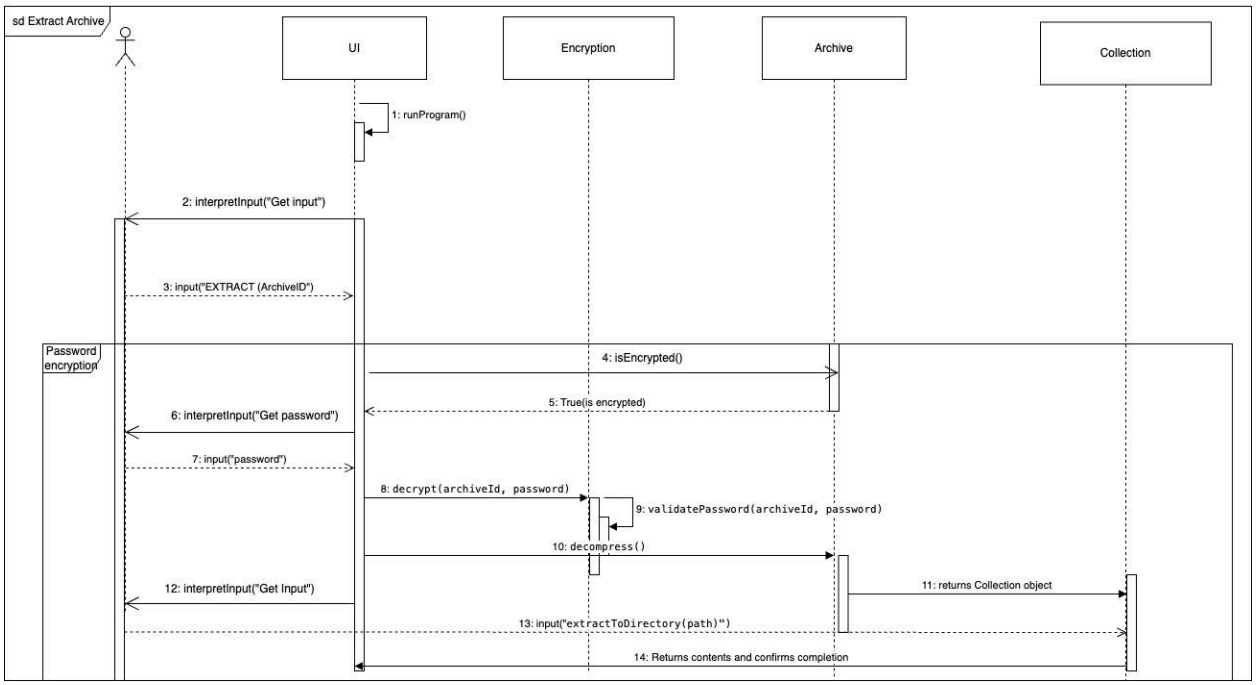


Figure 7: Sequence diagram of extracting an archive

The focus of this sequence is taking an existing compressed archive and extracting the files contained within it. Starting with the **UI (UserInterface)**, the user tells the **UI** they want to extract file(s) from an **Archive**, using the `"EXTRACT (ArchiveID)"` command. **UI** will then use *isEncrypted()* on the **Archive** class which returns the true or false. In this case the archive is password protected so the **UI** asks the user for the password using *interpretInput()*.

This is where the **Encryption** class checks if the password is correct, initiating the decrypting process **UI** calls *decrypt()*. **Encryption** will call *validatePassword()* on itself once again returning a boolean value. If the password is validated, **UI** then calls *decompress()* on the **Archive** class which returns a **Collection** object. Finally the user calls *extractToDirectory()* to finally return the extracted and decompressed archive.