Assignment 3: Implementation

Team 3

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.

Summary of changes from Assignment 2

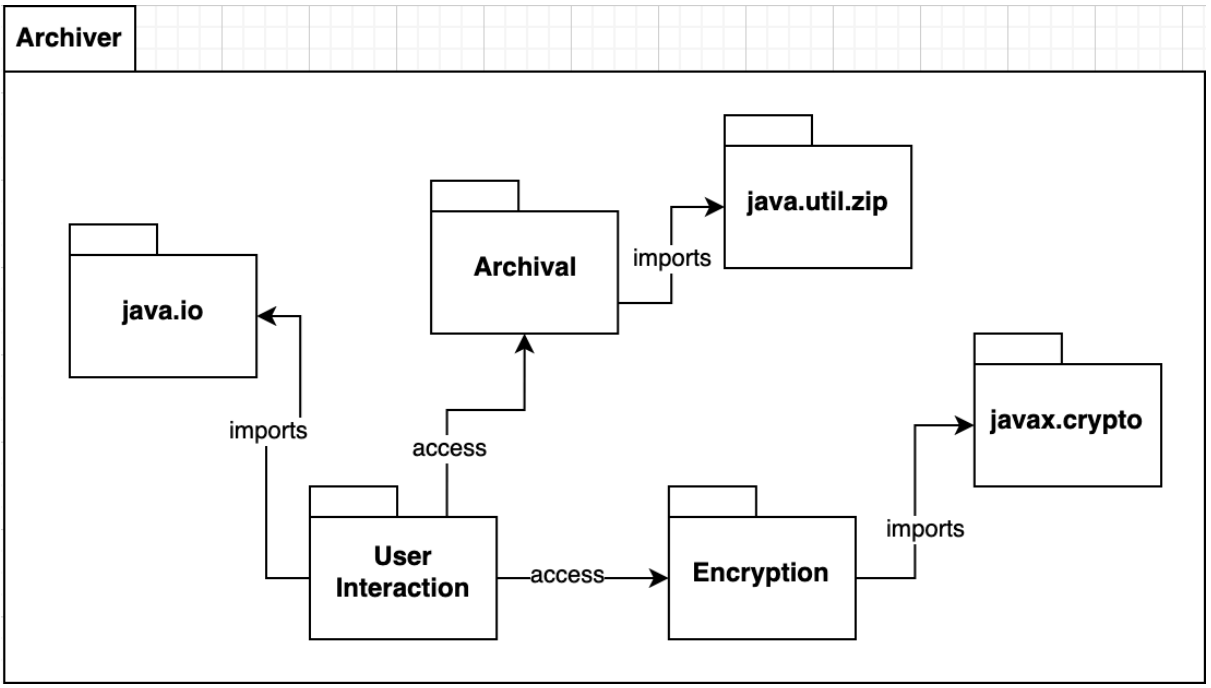
Author(s): Charlee, Ania, Luca, Jacob

Disclaimer: our implementation is not 100% functional. The compression and decompression features have been tested and are working. The methods in the **Encryption** class have also been tested separately, but we have been experiencing an error upon decryption that we worked tirelessly to fix but to no avail.
In general, please pay more attention to the classes themselves rather than how they are being called in **UserInterface**, as this is the class we were completing testing in.

- We deleted the **Collection** class and revised all the diagrams accordingly. During implementation, we found that **Collection** was a poor design as it added redundancy and had very shallow methods that merely made calls to other classes.
- **LZ4** was built but we could not successfully tie the logic to the framework of our Archival process and thus it had to be left disassociated for now.
- Added combined fragment(s) to sequence diagrams based on Assignment 2 feedback.
- Revised state machine diagrams according to Assignment 2 feedback.
- Changed the **Collection** state machine to **Encryption** state machine since the **Collection** class was deleted.
- We are no longer keeping track of a list of archives. We discovered that it is simpler to have the user select the desired archive from their local machine at each step of the process.

Revised package diagram

Author(s): Luca



Collection Removal

- Following previously mentioned decisions to remove the collection functionality due to redundancy, **Collection** was removed and its functionality given to **Archival**.

LZ4 Removal

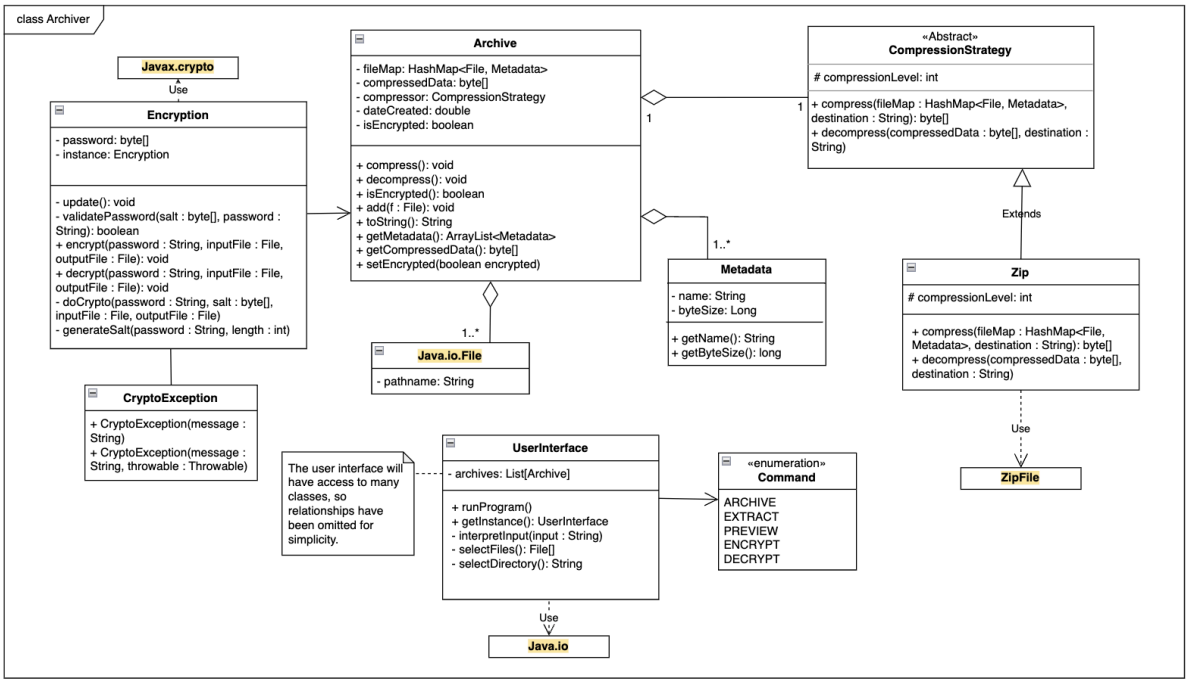
- Due to constraints, we could not fully implement **LZ4** compression and so this package was similarly removed.

Zip4J Removal

- Due to the fact that the built in zip functionality was more convenient and that the **Zip4J** package did not provide utility benefits that outweighed the bloat of importing another package, we stuck with the built-in **Java.util.zip** and removed **Zip4J** from our dependencies.

Revised class diagram

Author(s): Charlee



Archive

- Archives no longer have an ID, as we decided to not keep track of the archives on our program. Instead, the user can specify an archive by selecting a file from their local machine.
- Added method *toString()* to print metadata.
- Changed *previewContents()* to *getMetadata()*. We also no longer store the file type in metadata because it was a redundancy (file type is already included in the string name).

Collection

- We removed the **Collection** class entirely to simplify our program. It was acting merely as a transition between the files entered by the user and the **Archive** and did not add any new functionality.

Encryption

- Only one password will be stored because we are now assuming that only one archive will be created and encrypted during the execution of the program.
- *doCrypto()* and *generateSalt()* are the most important additions from Assignment 2, as we have now flushed out the details of what encryption requires logistically. *doCrypto()* can be used for both encryption and decryption depending on *cipherMode.*

CompressionStrategy/Zip

- We removed Lz4 as the implementation is not fully complete. However, if this were to be completed in the future, it would not require any major modifications to our system to include that functionality.

CryptoException

- We added this class to handle exceptions occurring during the encryption process. This is a good practice because it improves specificity of errors.

Application of design patterns

Author(s): Ania, Charlee, Jacob, and Luca

	DP1
Design pattern	Singleton
Problem	We wanted the **UserInterface** class to control the flow of the application. Throughout the execution, the user will be entering input and actions will be taken based on that. We wanted to ensure that only one instance of this class was created so that the state of the system can be smoothly tracked.
Solution	We have applied the Singleton design pattern by making the constructor of **UserInterface** private and creating a static method that acts as the constructor. This instance is also able to store any global variables where necessary.
Intended use	When the program first starts, a new instance of **UserInterface** is created. Any time this method is called after that, it returns the same instance.
Constraints	
Additional remarks	

	DP2
Design pattern	Singleton
Problem	A single instance of the Encryption object is required in the file archiver application to manage encryption and decryption operations effectively. Without the Singleton pattern, multiple instances could potentially lead to inefficiencies, resource duplication, and conflicts.
Solution	Implementing the Singleton design pattern ensures that only one instance of the Encryption object exists throughout the application's lifecycle. It simplifies access to encryption functionality and facilitates seamless integration with other components of the application.
Intended use	At runtime, when the file archiver application initializes, it instantiates the Encryption object using the Singleton pattern. Subsequently, any module requiring encryption or decryption services accesses this single instance of the Encryption object. This ensures that all encryption-related operations are consistently managed and that any updates or modifications to the encryption logic are applied uniformly throughout the application.
Constraints	One constraint imposed by the Singleton pattern is that it restricts the instantiation of the Encryption object to a single instance, which could potentially limit scalability or concurrent access in scenarios where multiple threads or processes

	require independent instances of the Encryption object. However, in the context of the file archiver application, where centralized control and consistency are paramount, this constraint is acceptable and even beneficial.
Additional remarks	The use of the Singleton pattern not only addresses the immediate need for centralized control over encryption operations but also contributes to the overall robustness, maintainability, and performance of the file archiver application. It promotes a modular design and facilitates future enhancements or extensions to the encryption functionality with minimal impact on existing code.

Revised object diagram

Author(s): Ania

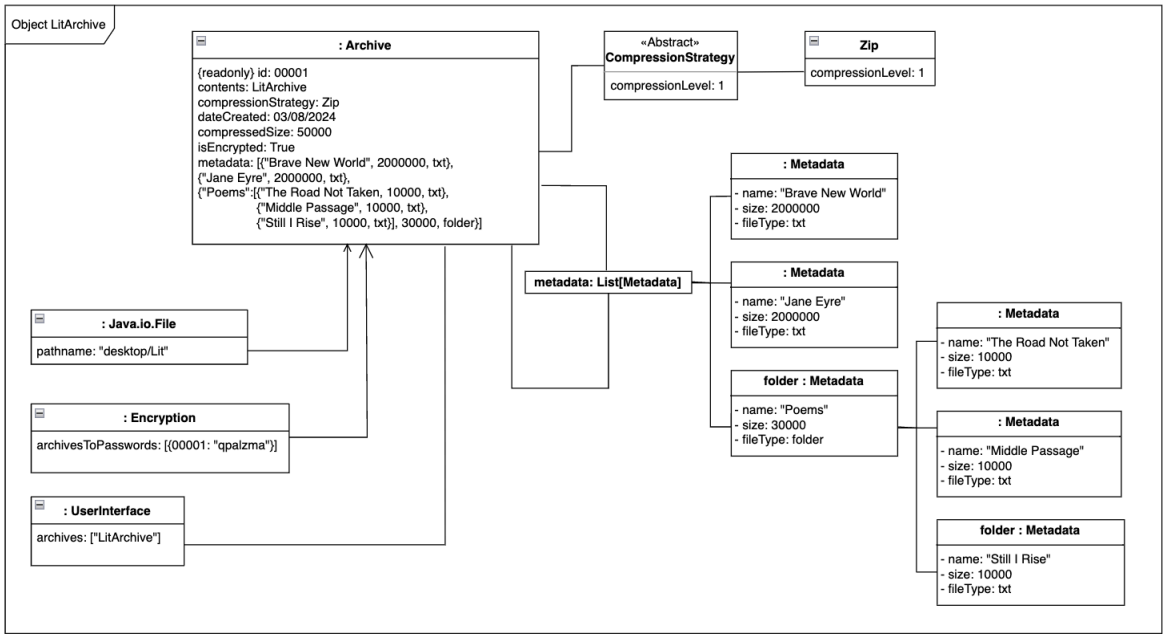


Figure 3: Object Diagram

The object diagram depicted in Figure 3 provides a comprehensive overview of our system's architecture, showcasing an **Archive** object to visually represent the relationships within the system. The process begins with the user initiating the archiving of a group of files, resulting in the creation of an **Archive** instance. 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.

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"}.

Furthermore, the **UserInterface** class maintains an object containing the list of existing archives, simplifying the management of archives within the system.

The object diagram underscores the Singleton pattern's application by emphasizing the existence of only one instance for both the Encryption and UserInterface classes. With **Encryption**, a single instance manages encryption and decryption operations, enhancing security and preventing conflicts. Similarly, the **UserInterface** class's Singleton instance ensures consistent interface management and user interactions.

In the evolution of our system, we made significant changes to streamline its functionality and eliminate redundancy. We removed the **Collection** class, as it became redundant after restructuring our architecture. Instead of first creating a **Collection** object from specified files and then archiving it, the files are now directly processed by the **Archive** class. This optimization enhances the efficiency and simplicity of the archiving process.

Overall, the changes made to our system's object diagram reflect our commitment to enhancing performance, usability, and maintainability, while also addressing evolving user requirements and design considerations.

5 - Revised state machine diagrams

5.1 Archive class

Author(s): Ania

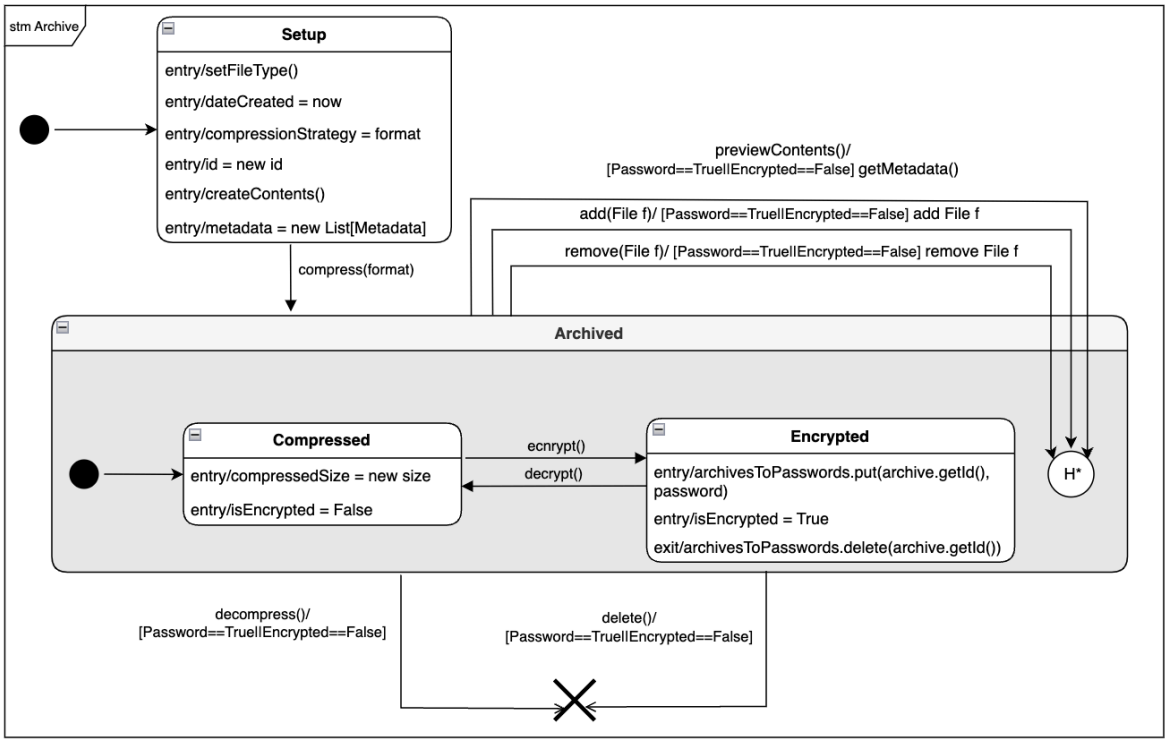


Figure 4: State Machine Diagram of the Archive Class

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

To facilitate the *getMetaData()* function when archiving the files, we have chosen to duplicate metadata containing details such as the name and size of every file in the archive, 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.

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 and extraction of files, resulting in the termination and deletion of the archive. 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. *getMetaData():*
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 *getMetaData()* function makes use of the metadata that was created before archival. If the archive is not encrypted, *getMetaData()* functions seamlessly. In the event of an incorrect password, no action is taken.

3. *add(File f):*

When the user invokes the *add(File f)* method, the program prompts for the encryption password if the archive is encrypted. Upon providing the correct password, the program securely adds the specified file, 'f', to the archive. This process involves encrypting the file (if necessary) and appending it to the archive. All associated metadata and references are updated accordingly. If the archive is not encrypted, the file is added directly. In case of an incorrect password, the file addition process is aborted, preserving the integrity of the archive.

In our design of the **Archive** class, we've made deliberate choices to ensure clarity and functionality in representing the states and transitions. We've structured the states, starting from ***Setup*** and progressing through to the ***Archived*** composite state, which encapsulates both ***Compressed*** and ***Encrypted*** states. This approach provides users with a unified view of the archive's different states and functionalities, streamlining the state machine for easier

comprehension. By including specific details about each state's functionalities and transitions, such as password prompts for encryption and decryption, we aim to facilitate a smooth user experience and clear understanding of the **Archive** class's behavior. Additionally, we've prioritized user convenience by implementing the *previewContents*() function, which relies on copied metadata for efficient previewing of archive contents without decompression. Overall, our design choices strike a balance between clarity and functionality, offering users a straightforward yet comprehensive understanding of how the **Archive** class operates.

5.1 Encryption class

Author(s): Ania

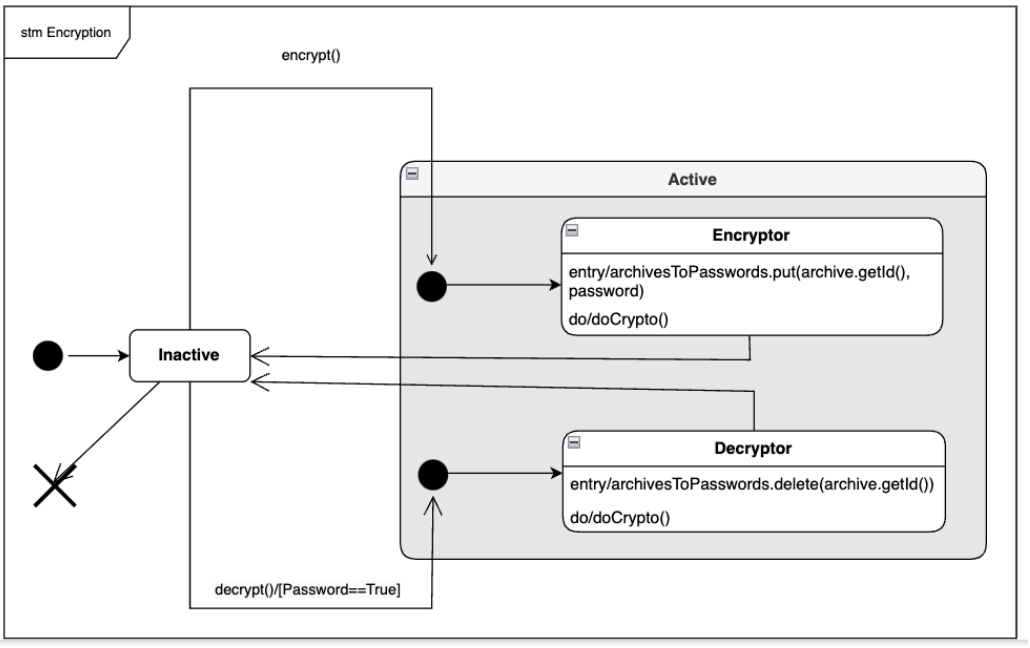


Figure 5: State Machine Diagram of the Encryption Class

The **Encryption** class represents a state machine governing the behavior of encryption operations within the system. Upon instantiation, the **Encryption** object enters the ***Inactive*** state, adhering to a singleton pattern to ensure only one instance exists within the system. In this state, the **Encryption** object remains idle, not performing any actions.

Transitioning from the inactive state, the user may invoke the *encrypt*() method, prompting the object to transition to the ***Active*** state with the ***Encryptor*** substate. During this transition, the **Encryption** object records the provided password and archive ID in a secure data structure such as a hashmap, while internally executing the encryption process via the *doCrypto*() method. Upon completion of encryption, the object reverts to its inactive state.

Conversely, if the user initiates decryption by calling *decrypt*(), the object transitions to the ***Active*** state with the ***Decryptor*** substate. The guard [Password==True] verifies the correctness of the provided password before proceeding. Subsequently, the password is removed from the hashmap storing encrypted archives and their passwords, and decryption is performed through the *doCrypto*() method. Upon successful decryption, the ***Encryption*** object returns to its inactive state.

The state machine persists throughout the application's lifecycle, terminating only upon shutdown. The design choice to enforce a singleton pattern ensures centralized control over encryption operations, preventing multiple instances from conflicting or redundant actions. The use of substates (***Encryptor*** and ***Decryptor***) encapsulates the specific functionalities of encryption and decryption, promoting modular and maintainable code. Additionally, the decision to retain the ***Encryption*** object throughout the application's runtime enhances security by preventing inadvertent deletion or tampering.

Revised sequence diagrams

Author(s): Jacob

Creating an Archive

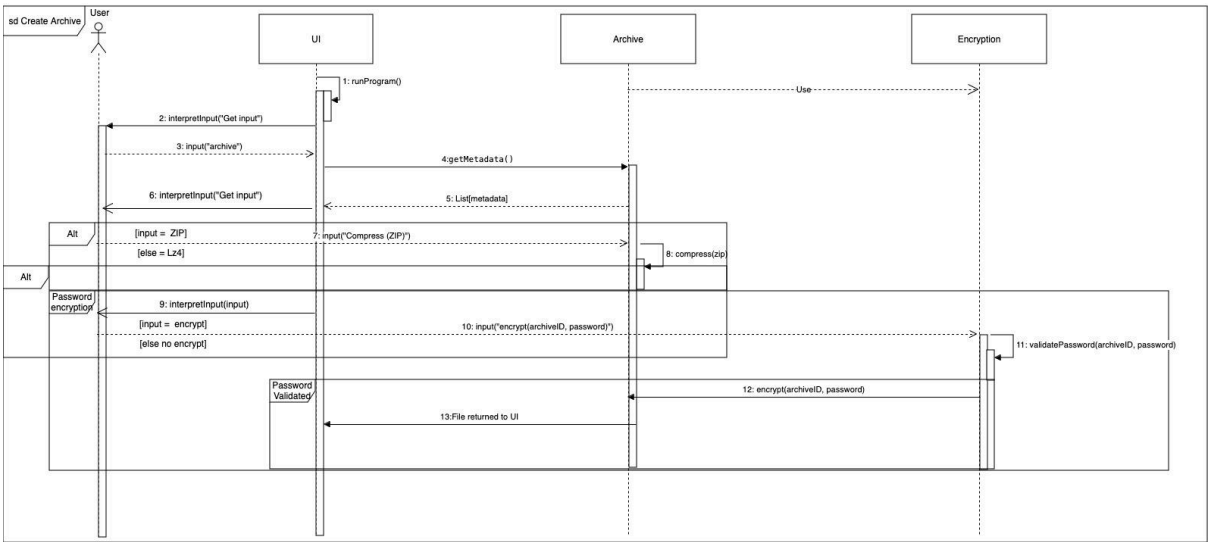


Figure 6: Sequence diagram of creating an archive

In this sequence, we are showing the process of creating a compressed **Archive** from a group 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 **Archive** 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

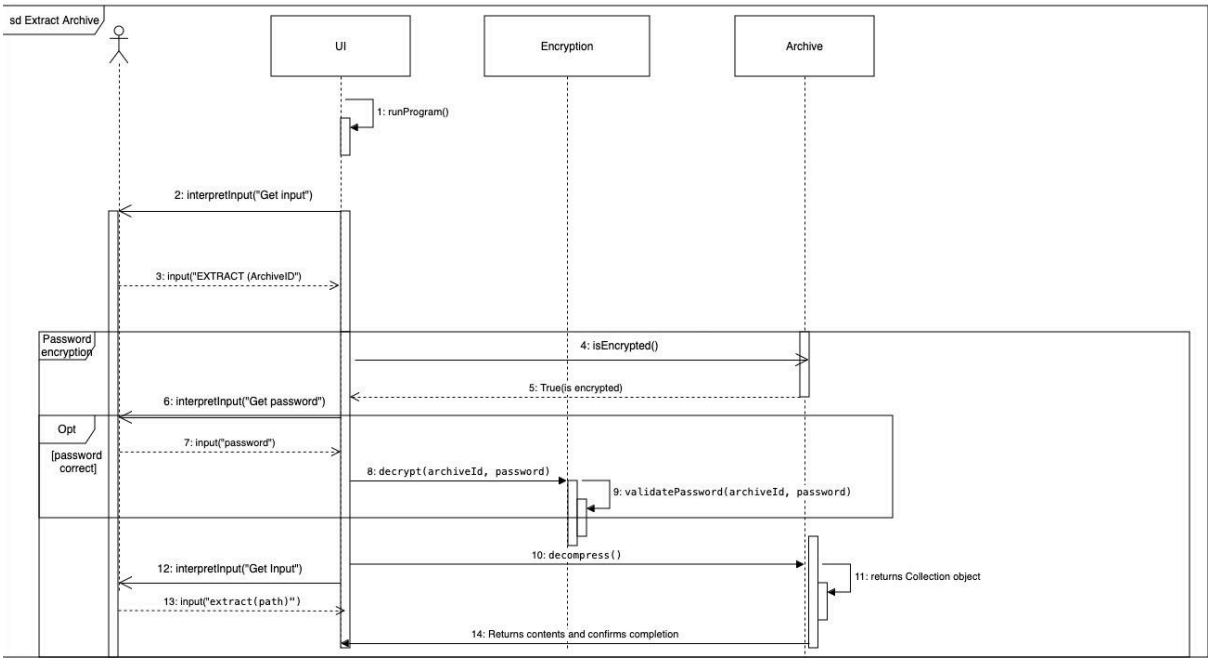


Figure 7: Sequence diagram of extracting an archive

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 the decompressed file(s). Finally the user calls *extractToDirectory()* to finally return the extracted and decompressed archive.

Implementation

Author(s): Ania, Charlee, Jacob, and Luca

In this chapter, you will describe the following aspects of your project:

- the strategy that you followed when moving from the UML models to the implementation code;

- the key solutions that you applied when implementing your system, e.g., for especially challenging functionality or functionality where a lot of different options existed for the implementation;
- the location of the main Java class needed for executing your system in your source code;
- the location of the JAR file for directly executing your system;
- the link to the 30-seconds video showing the execution of your system (you are encouraged to put the video on YouTube or the video platform of your choice).

IMPORTANT: remember that your implementation must be consistent with your UML models. Also, your implementation must run without the need to access any other externally running software. Failing to meet this last requirement means 0 points for the implementation part of your project.

Maximum number of pages for this section: 4

Time logs

<Copy-paste here a screenshot of your time logs - a template for the table is available on Canvas>

Member	Activity	Week number	Hours
All	Initial team meeting	2	1
Luca Snoey	Assignment 1	2	1
Charlee Lachance	Assignment 1	2	1
Ania Serbina	Assignment 1	2	1
Jacob Roberts	Assignment 1	2	1
All	Mentor meeting and assignment 1	2	1
All	Peer feedback	3	1
All	Mentor meeting reviewing assignment 1	3	1
All	Team meeting to dicuss assignment 2	4	2
All	Mentor meeting about assignment 2	4	1
Charlee Lachance	Assignment 2	4	2
Luca Snoey	Assignment 2	4	2
All	Team meeting to dicuss assignment 2	5	1
Charlee Lachance	Assignment 2	5	6
Ania Serbina	Assignment 2	5	6
Jacob Roberts	Assignment 2	5	6
All	Mentor meeting for assignment 2	5	1
Luca Snoey	Assignment 2	5	3
Charlee Lachance	Assignment 3	7	7
Ania Serbina	Assignment 3	7	7
Jacob Roberts	Assignment 3	7	7
Luca Snoey	Assignment 3	7	7
Charlee Lachance	Assignment 3	8	7
Ania Serbina	Assignment 3	8	7
Jacob Roberts	Assignment 3	8	7
Luca Snoey	Assignment 3	8	7
		TOTAL	94