Assignment 2

Team number: 57
Team members: Dennis Moes, Simon Vriesema, Joli-Coeur Weibolt en Jaïr Telting

Name	**Student Nr.**	**Email**
Dennis Moes	2839746	d.moes@student.vu.nl
Joli-Coeur Weibolt	2837627	j.r.f.weibolt@student.vu.nl
Simon Vriesema	2839785	s.r.vriesema@student.vu.nl
Jaïr Telting	2691376	j.s.student@vu.nl

Format: establish formatting conventions when describing your models in this document. For example, you style the name of each class in bold, whereas the attributes, operations, and associations as underlined text, objects are in italics, etc. Consistency is important!

Content

Summary of changes from Assignment 1

Author(s): Joli-Coeur Weibolt

Defining the interface:
Initially, in our previous overview we gave a basic description in the form of bullet points of what our file archive will entail. Our TA mentioned in our feedback and the weekly meeting to give more details about the user experience and what for type of interface we want to make. Based on the given feedback we made sure to clearly state that we are going to make a GUI. We also gave a detailed explanation of how the user would be able to use the file archive.

Specifications:
In our TA's feedback there was a lot of comment at specifications and defining requirements. Most of the features and requirements we stated where not detailed enough explained. With our current version we made sure that:

- The extraction settings are more specific
- The standards we are referring to are more specific
- The optimal compression format is defined

Deletions:
The deletions we made are based on the given feedback of our TA, we removed the examples we had on feature 6 and replaced them with a list. We removed the examples, because it does not belong to the basic conventions when stating a requirement. We also completely removed feature 7 – Independence from Compression Formats, because the way it was formulated was not following the basic conventions at all. The only requirement of feature 7 was already stated in feature 6. We removed 'file details' from feature 4, because we thought file details and metadata was redundant.

Package diagram

Author(s): Dennis Moes

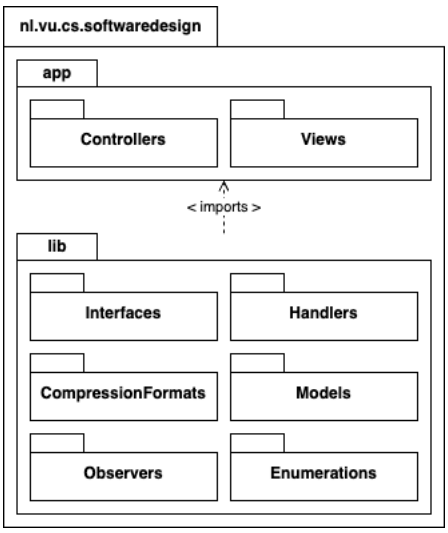


Figure 1 - Package Diagram

In Figure 1 we can see our diagram for the structure of our application. The design emphasizes a difference between the user interface (UI) and the backend code which will do the heavy lifting. This made us create two separate packages: “app” and “lib”.

Within the “app” package, we will put all the code that is responsible for managing the UI. This means that we will put the various views and controllers associated with the application in it. The controllers will be what interact with the views.

The “lib” package will serve as our backend. Most of the classes outlined in Figure 2 will find their place within this package. This includes the interfaces, compression formats, models, observers, and enumerations. To make sure that every file has a package, we made a “Handlers” subpackage. The “Handlers” subpackage acts as a catch-all for the classes that we couldn’t easily put in a separate package.

Class diagram

Author(s): Dennis Moes, Joli-Coeur Weibolt, Jaïr Telting, Simon Vriesema

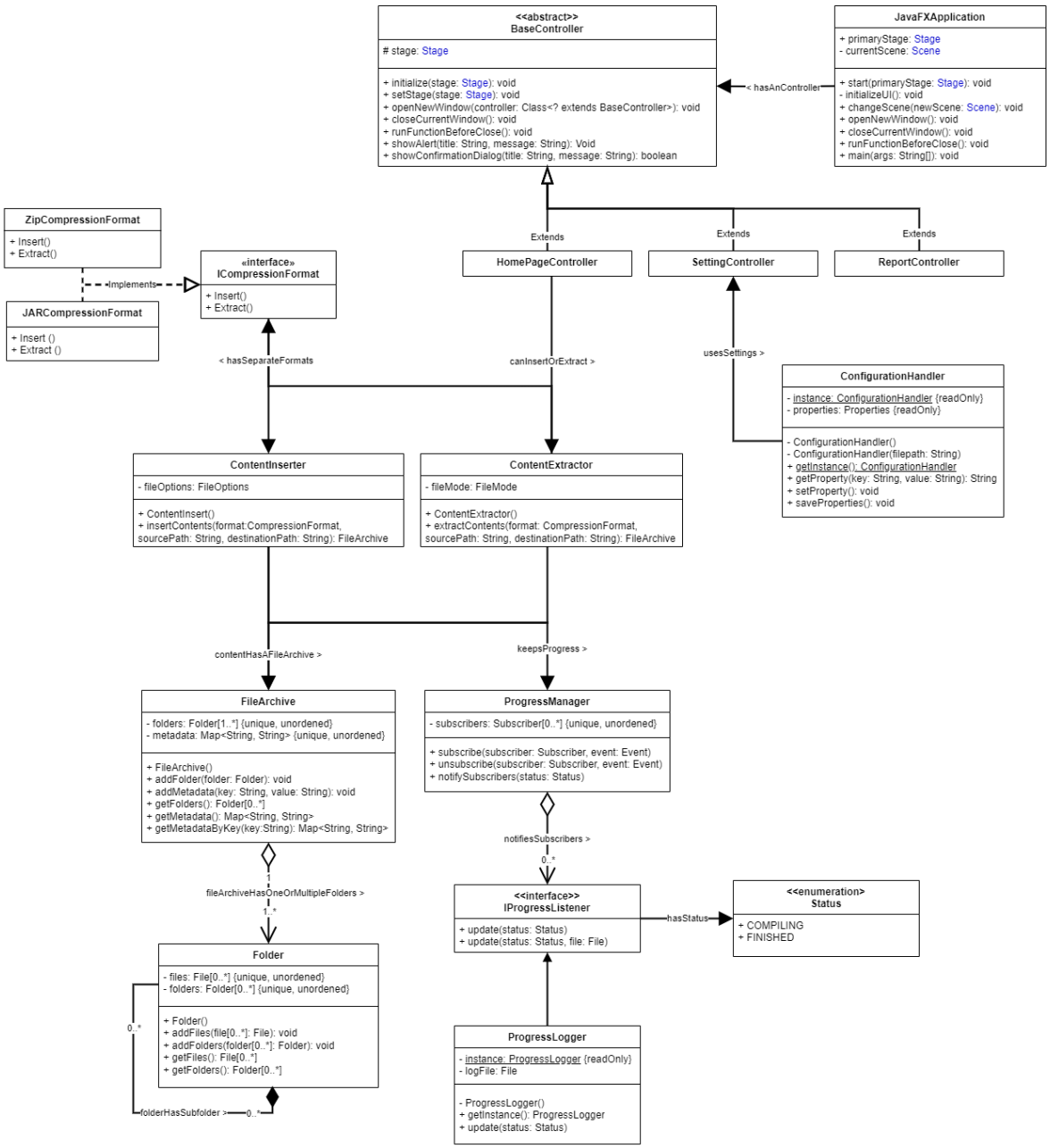


Figure 2 - Class Diagram

Note: we omitted the JavaFX fields and methods in this diagram to keep focus on the inner class structure and avoid polluting our diagram with a large controller.

Singleton classes

ProgressLogger: With the logger class we want to write our logs into one log file. This class will open and use one file. Because we don't want to open multiple instances of one file and have an error later one we created a singleton class out of it which means it will have only one instance and ergo only one instance of the opened file.

ConfigurationHandler: Just as the ProgressLogger we want to have only one configuration file open and in use at any time. This is because we want to use the same settings across the whole application. This file will open the configuration file and store the configurations locally in a variable. If we want to read or write any configurations it will firstly read or write to the local variable and then the user or application can choose to save it which will store it in the one configuration file.

Interfaces

ICompressionFormat: To enable multiple compressionformats we created aICompressionFormat interface in which we defined the functions of insert and extract. This means we can more easily add compression formats in our application. The new compressionformat would implement ICompressionFormat and then every new compressionformat would take in the function from ICompressionFormat and override the Insert and Extract function to its own functionality.

IProgressListener: This interface describes what functions the listeners in the observer would need. In our case every listener needs an update function. This tells the ProgressManager what every listener can do. In our case it needs the update.

Observers

ProgressManager: To enable automatic logging we want to implement a Notifier which notifies to all its subscribers. In the case of this diagram (it could be extended in assignment 3) we want to have the ProgressLogger to listen to the notifier ProgressManager. For example if we create an archive we would send a notification via the ProgressManager to each listener. And then the ProgressLogger would log to the log file.

Enumerations

Status: This enumeration tells the application what status the archive is in. This will help the logger know what to log into the log file. For example If the archive is compiling it will write compiling and if it is completed it will write completed in the log.

JavaFX controllers

BaseController: This will be our abstract class which all controllers will extend from. In this class we put the most important code which needs to be shared with all our sub controllers. It has a protected Stage variable which can be used to move to another controller. It also has functions to show an alert, yes-no dialog or even to open a new popup window. In this class we also have the important function of runBeforeClose. This will run some code before the application can close such as autosaving or checking if an archive is completed and waiting for it to finish.

Regular Classes

FileArchive: This class is one of the most important if not most important class in our system. This File Archive keeps track of all our available files in an archive. How does it do that? It keeps track of all the Folders there are in an archive with the Folder class. So, let's say you want to add a file to your archive. The user will select the file it wants and the system will automatically find out to which folder it belongs to. If the user wants to it can also add some metadata to the archive, such as a password, a text or even a secret message.

Folder: As mentioned in the FileArchive description, this class is a part of the FileArchive class. And it is a sort of a recursive class. We remember in this file which files we need for the archive and which subfolders we also need. It is a sort of tree data structure. And as for the FileArchive if we add a file to it the system will dynamically know where to put it in which subfolder. This will also make sure that we can keep the original structure of before the archiving.

Object diagram

FileArchive

Author(s): Dennis Moes

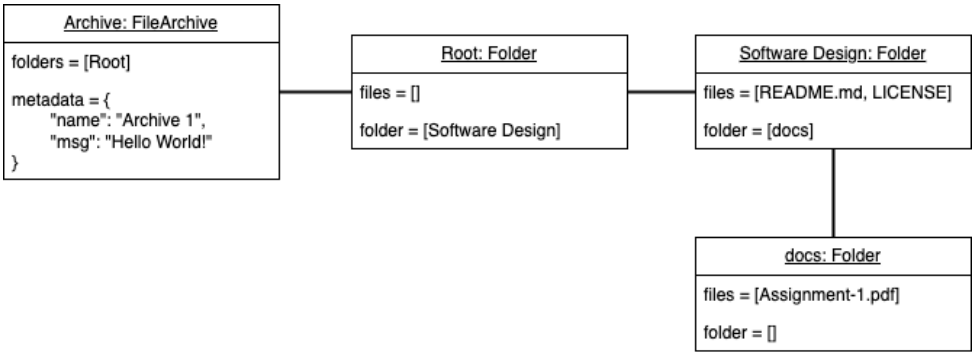


Figure 3 - Object Diagram

In Figure 3 we can see an object diagram of an instantiated **FileArchive** along with it's subfolders. This gives us a view into the. Hierarchical structure within the archive. Within the first container we see an **FileArchive** instance called *Archive*.

The *Archive* class instance stands as the top-level container, this top-level container holds the metadata associated with this archive. This metadata uses a key-value pair, this makes it a versatile system which can accommodate a diverse range of metadata. In this case we added a name and a message to the archive.

Associated with the *Archive* is the **Folder** labelled *Root*. This folder is important as it is the starting point for organizing the archive's contents, this serves as the reference point for the overall structure. To add demonstrate that there can be nested folders there is another folder attached to it called *Software Design*. This nested folder is used to house files we want to archive such as README.md and LICENSE.

Because of this nested structure in the instantiated class, we can keep the nested structure in the archive. When we want to archive a folder, we pass the *Archive*. This will show the application in which structure we will keep the archive. Also, with the *Archive* we added some extra information such as the name and an optional message linked to this archive.

State machine diagrams

ConfigurationHandler:

Author: Simon Vriesema

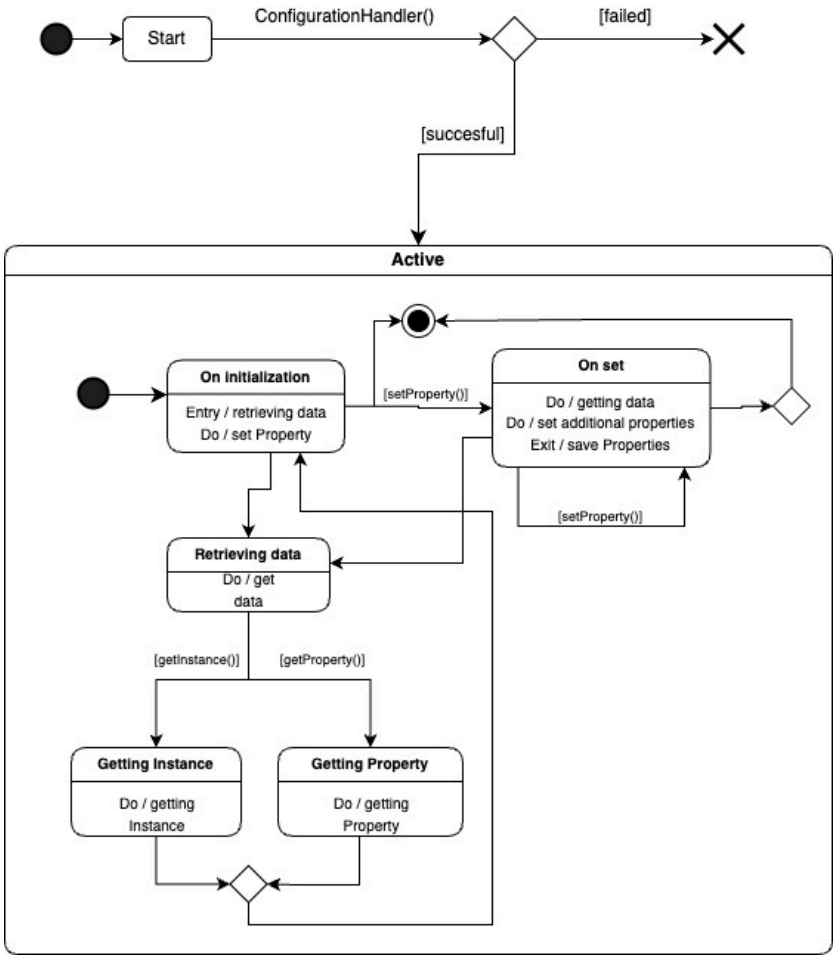


Figure 4 - ConfigurationHandler State Diagram

When an object of the ConfigurationHandler class is created successfully, it transitions into the active state. In the active state, users have the ability to perform various operations. They can retrieve data using the getInstance() and getProperty() methods. Additionally, users can set properties using the setProperty() method. At any point during the active state, users can continue setting properties or retrieving existing ones.

Once users are satisfied with their modifications, they can invoke the saveMethod() function to persist the changes. Upon calling saveMethod(), all modifications made during the active state are saved. After saving the changes, users have the option to finish this session, transitioning out of the active state.

Class diagram FileArchive

Author: Jaïr Telting

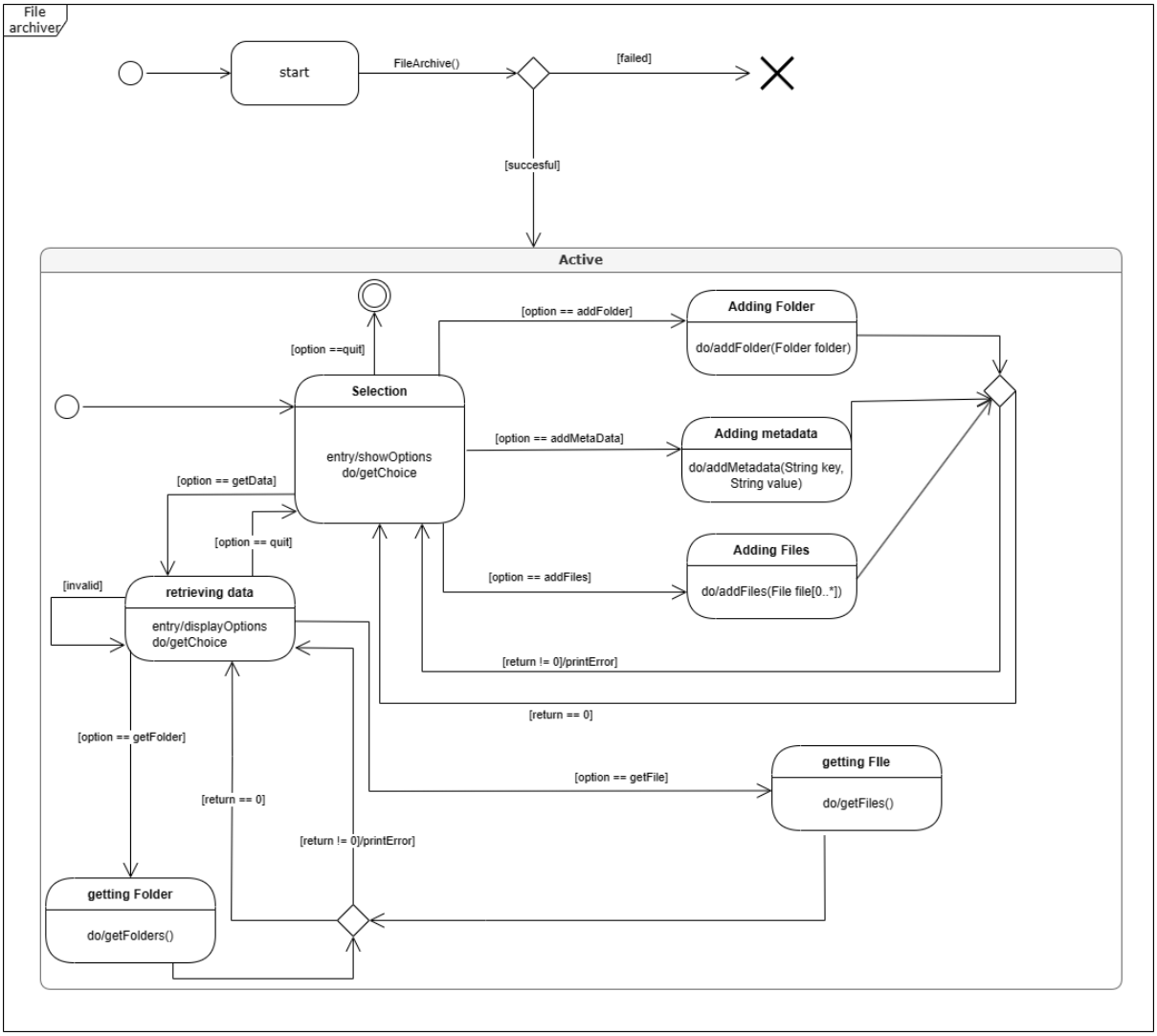


Figure 5 – State Diagram of the FIleArchive class

When creating an object of the class it enters the starting state. When the file archiver successfully starts, the class enters the "active" state.
In the "Selection" state, the entry activity "show options" is performed. Display the possible interaction options the user can perform" is performed. The interaction choices are:

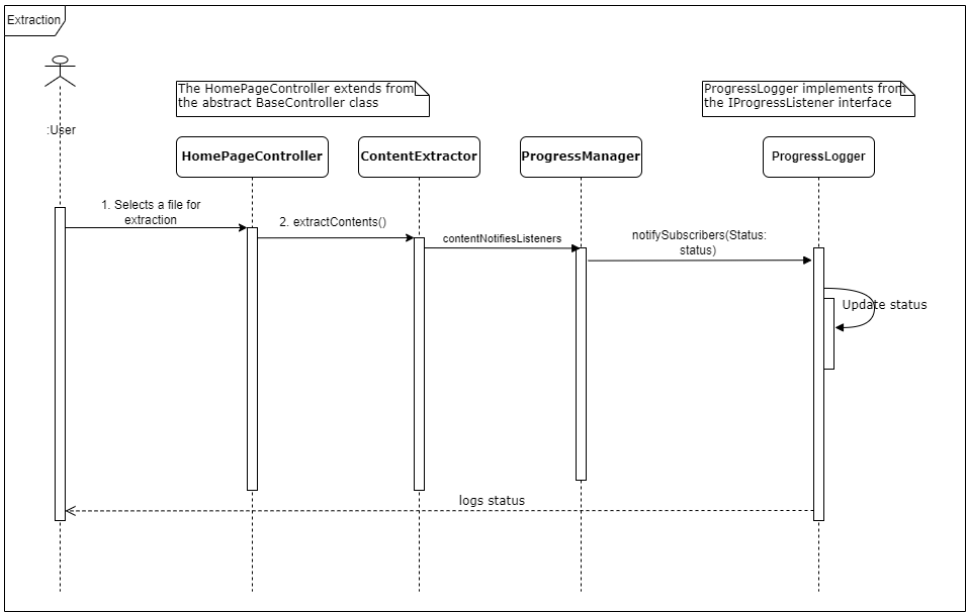
1. Add folder. In this case, the class enters the "Adding Folder" state.
2. Add a file. In this case, the class enters the "Adding File" state.
3. Add metadata. In this case, the class enters the "Adding Metadata" state.
4. Show data. In this case, the class enters the "retrieving data" state. Additionally, in this state, the entry activity "displayOptions" (which is similar to the "showOptions" entry activity) is performed. The interaction choices for this state include:
    a. Get a folder (show a folder). In this case, the class enters the "getting folder" state.
    b. Get a file (show a file). In this case, the class enters the "getting file" state.
    c. Invalid. In this case, the class re-enters the "retrieving data" state.
5. Exit. In this case, the class enters the final state.

After the internal activities following the selection made in the "selection" and "retrieving data" state finish, the class returns to their respective state. Afterward, the user can select another interaction.

Sequence diagrams

Author(s): Joli-Coeur Weibolt

Extraction sequence



The sequence diagram above shows the order of our file extraction. We start with the user, the user sees a GUI and can decide to export a file. From the user a call is made to the HomePageController. The HomePageController is a controller that is an extension of the abstract BaseController class, which has a lot of different operations. Then the HomePageController calls the extractContents function of the ContentExtractor class. We thought it would be a better idea to split import and export to keep it simpler. The contentExtractor class then communicates with the progressManager to start the extraction. The progressManager is an observer and is responsible for the progress of the extraction. The progressManager calls the ProgressLogger using the notifySubscribers operation and The Progresslogger class extends from the IProgressListener interface and can update the status.

As the extraction progresses, the ProgressLogger updates the status and logs it back to the User. By implementing the ProgressLogger we can give the user real-time feedback about the extraction.

This sequence diagram highlights the collaboration between our different components and shows a clear design for our file extraction. Overall, this sequence provides a user-friendly experience by keeping the user informed about the progress of the file extraction.

Sequence Diagram Adding data

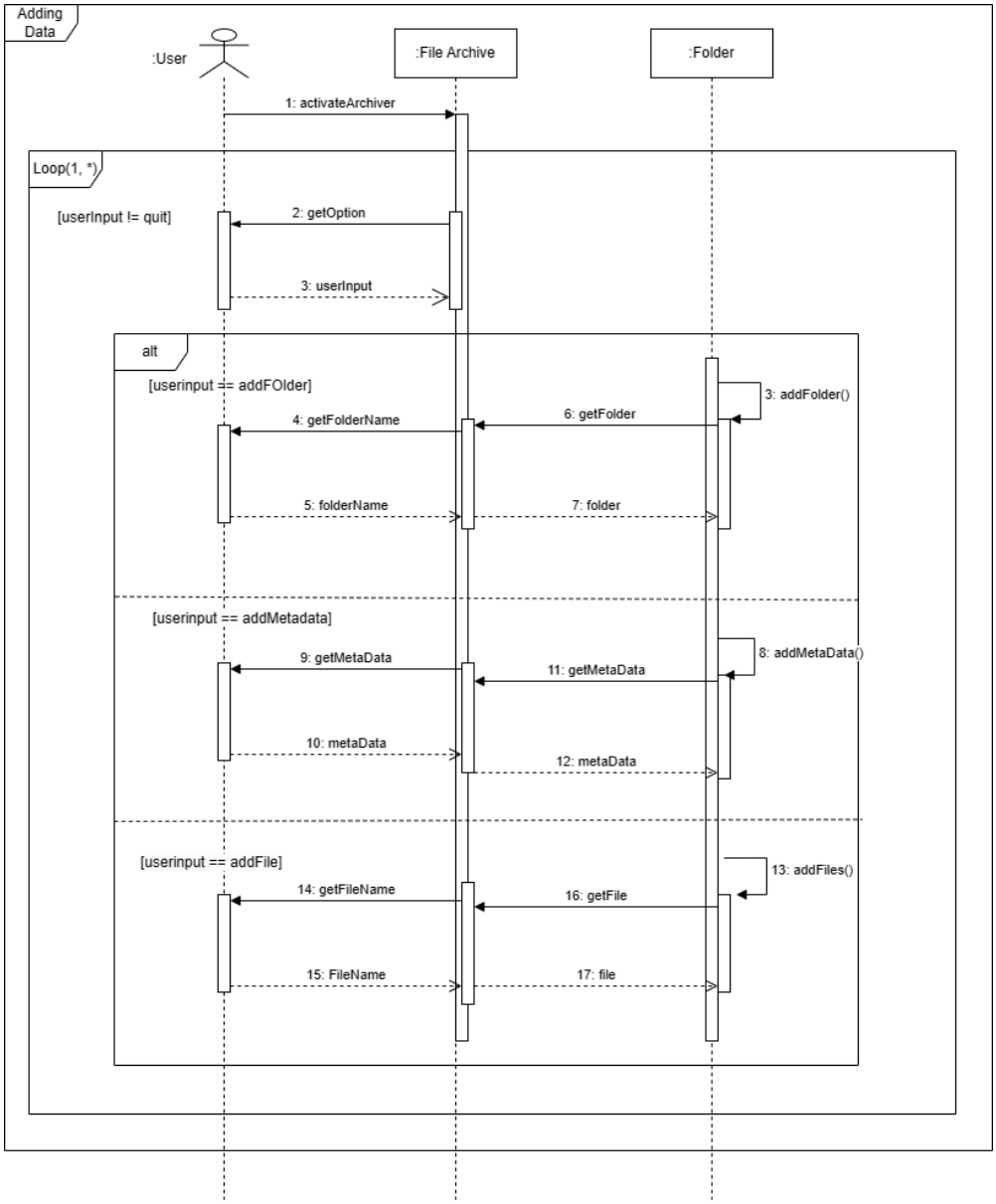


Figure 6 – Sequence Diagram of adding data

The sequence diagram depicted in Figure 6 displays shows the communication sequence between a user and instances of the FileArchive and Folder Class. The interaction starts with the user activating the file archiver (FileArchive). Activating the file archiver enables the loop of the FileArchive class, where the user is prompted to enter their action. This triggers the interaction FileArchive - User by the getOption() function. The result is stored in the user input variable. There are four possible valid inputs, resulting in four possible execution paths: add a folder, add metadata, add files, and exit. When the exit option is selected by the user, the loop is exited. The remaining three options the User can choose from are displayed by an alt fragment.

The first operand represents the alternative path for when the user wants to add a folder (user input == addFolder). This triggers the function addFolder(), which needs to acquire the

folderName(through getFolderName). Afterward, FileArchive creates a new instance of Folder.

The second operand represents the alternative path for when the user wants to add metaData (user input == addMetaData). This triggers the function addMetaData(), which needs to acquire the metadata (through getMetaDataget). Afterward, FileArchive creates a new instance of Folder.

The third operand represents the alternative path for when the user wants to add a file(user input == addFile). This triggers the function addFile(), which needs to acquire the fileName (through getFileName). Afterward, FileArchive creates a new instance of Folder.

This chapter contains the specification of at least 2 UML sequence diagrams of your system, together with a textual description of their elements. Here, you have to focus on specific situations you want to describe. For example, you can describe the interactions during a key use case of the software, during a typical execution scenario, in a special case that may happen (e.g., an error situation), etc.

Time logs

Team number	57			
Member	**Activity**	**Week number**	**Hours**	
Dennis Moes	Designing feature 1, 6 and the JavaFX controller class	3 & 4	4	
Joli-Coeur Weibolt	Designing feature 4, 5, extraction sequence diagram, summary assignment 1	3 & 4	8	
Simon Vriesema	Designing feature 2, 3 and the state machine diagram for ConfigurationHandle	3 & 4	6	
Jaïr Telting	Designing feature 7, sequence adding data and state diagram File archive	3 & 4	10	
		TOTAL	28	