Assignment 2

Team number: 99
Team members

Name	Student Nr.	Email
Bence Matajsz	2762173	b.matajsz@student.vu.nl
Thijmen Verlaan	2769118	t.r.verlaan@student.vu.nl
Ana-Maria Musca	2757719	a.musca@student.vu.nl
Yasin Karyagdi	2745681	y.k.karyaaedi@student.vu.nl

Summary of changes from Assignment 1

Author(s): Yasin

Feature 1:

- We now specify that the default format type will be ZIP.

Feature 2:

- We now specify that if the user does not define a folder to extract into, the location of the newly created folder will be in the current working directory.

Feature 3:

- We removed the usage of the -p flag when extracting and changed it so that the user is prompted for a password when trying to extract an archive that has been created with a password.
- We now mention that we are going to use an existing encryption algorithm, namely AES.

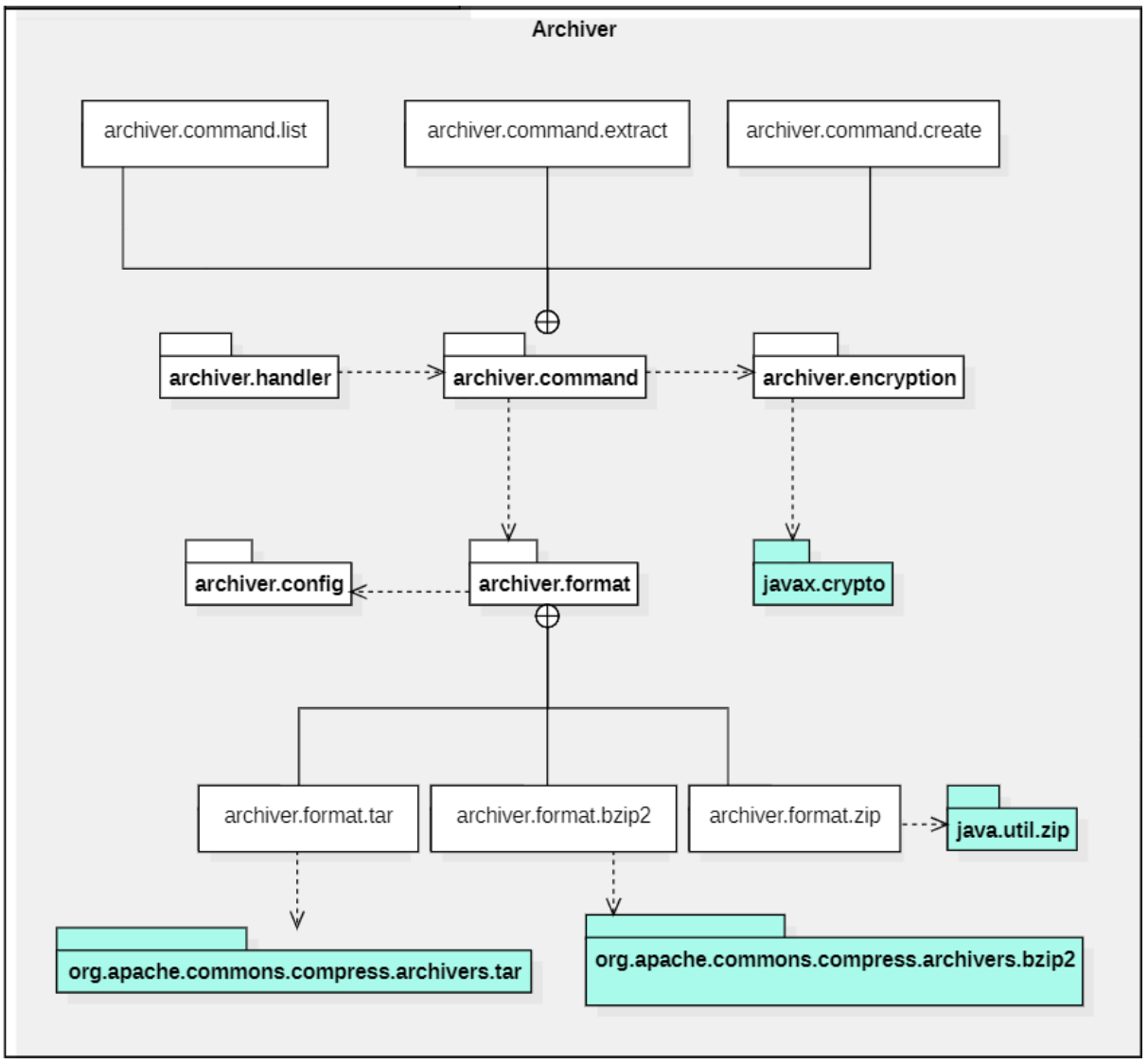
Feature 5:

- We changed it so that it's clear that the compression format is set during compression (so when creating in feature 1).
- We now mention which compression formats we are going to implement, namely ZIP, tar, and bzip2.

Package diagram

Author(s):Yasin

The diagram below shows the packages of the application to be built.



The classes are grouped in the following sub-packages of the package “archiver”:

- **Config:**
  This package contains the class Config, which we identified to be a separate component in the system. We could have incorporated it in the Format class but decided against it, since we thought that this separation allows the system to easily be extended in the future. Later on, configurations might be used elsewhere other than for compression. By doing it this way, we ensure that after you extend the Config package/class you can simply import the Config package in the new relevant component. As opposed to incorporating this part in the Format package/class, which would remove the ability to import configuration functionality without the addition of the format functionality.

- **Format:**
  This package contains the base class Format and the sub-classes Tar, Bzip2 and Zip. This package groups all the functionality of compressing and decompressing and does so with the usage of the abstract class Format. Each format extends this class and has to implement the corresponding abstract methods. This way we abstract away the specifics of how decompressing and compression works for each individual format, thus allowing the user to use the same familiar methods in order to do some operation without having to worry about the specific format in use. It furthermore allows new formats to easily be added to the project.

  This package imports the Config package due to compression being reliant on the configuration information. Additionally, each individual format we implement imports an external package which we will use in order to compress and decompress the archives. The external packages can be identified by their bluish green colour.

- **Encryption:**
  This package contains the class Encryption and groups the functionality of encrypting, decrypting and password handling. The package imports the external package “javax.crypto” in order to make use of its encryption and decryption methods, specifically for the AES algorithm. Again, the external package can be identified by its bluish green colour.

- **Command:**
  This package contains the base class Command and the sub-classes Create, Extract and List. This package groups all the functionality relevant to executing each command, which is done with the usage of the abstract class Command. Each command extends this class and has to implement the corresponding abstract method. This way we abstract away the specifics of how each command runs, which would additionally allow new commands to easily be added. This package imports the Format and Encryption packages due to needing their functionalities in order to execute the commands.

  Initially we wanted one big class Archive which would have consisted of a big class containing all the methods and attributes of the three classes previously mentioned. We decided against this design choice, since it did not make sense as to why you would need certain attributes and methods in cases where you will never end up using them. Thus we decided to split them into their own classes and group them within a package.

- **Handler**:
  This package contains the class Handler and groups the functionality of input/output. This is the package which contains the implementation of the command line interface. This package imports the Command package in order to get access to the commands it needs to run.

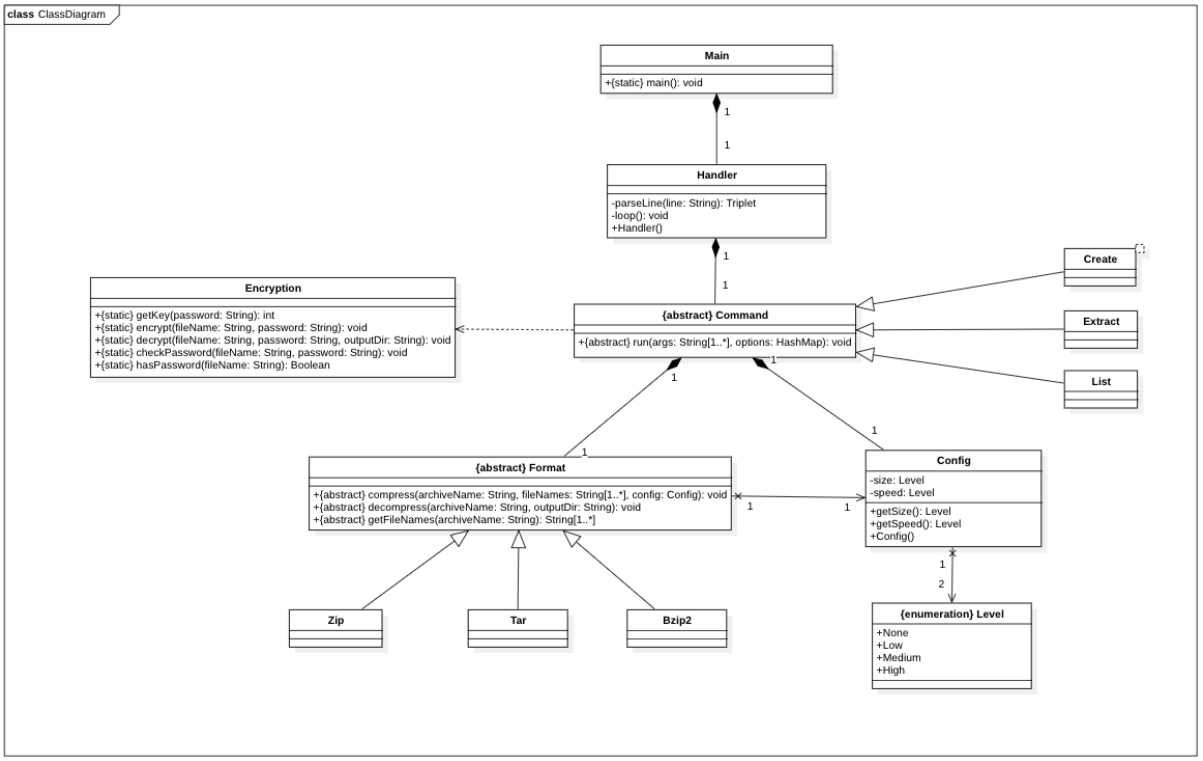
  We could have excluded this class and thus this package by implementing the input output handling and parsing in Main, but decided against doing so since we would like to keep the functionality out of Main. This way, any new user of the application does not have to worry about implementing their own Main and can

simply import the handler and run a working application. It additionally adds clarity regarding where the whole program starts its execution.

Class diagram

Author(s): Bence

The diagram below shows all the different classes to be implemented.



Data types:

- **Level:**
  This is an enum that specifies the levels that the attributes of a Config class can be set to. It contains four different values, including None, Low, Medium, and High. This data type has a directed association with the Config class, which is explained where the Config class is described.

Abstract classes:

- **Format:**
  This abstract class is the base for the different compression formats used throughout this project. It has three abstract methods that must be implemented by the subclasses that extend this abstract class. The first one is compress, which takes the names of the files that are to be compressed, as well as the configuration for the operation. This executes the format-specific compression of the provided files. The next one is decompress, which takes the name of the archive to be extracted and an output directory, and extracts this archive into the given output directory. The last method is getFileNames, which takes the name of the archive and returns the files stored in it. All subclasses will implement their own version of these three methods, and will be created upon executing the run method of Command.

- **Command:**
  This abstract class stands for the valid command the user may enter. The Handler class will create one of this abstract class's subclass, based on which command is being processed. It only has one abstract method, which every single subclass must implement in order to fulfil their functionality. This is the run method, which takes the arguments and options given to the specific command the user typed in. The correct subclass will be created and will execute their own run method. For example, the process of compressing. There is a one to one composition between this and the Handler class, as a Command subclass object can only exist if it is created in Handler. Furthermore, the Encryption class is a dependency for the Command class, because to successfully execute the tasks during the run method, it needs the methods of the Encryption class.

Static classes:

- **Encryption:**
  This class handles the encryption and decryption of the archives and provides the sufficient methods to successfully complete these tasks. The encrypt method takes the name of the file to be encrypted and a password to do so, then encrypts the archive with the provided password. Decrypt takes the same arguments and an additional string, which is the directory name where the archive should be extracted to. This method decrypts the archive. The method checkPassword takes the name of the archive and a password as arguments, and tests if the given password is correct for this archive or not. The class includes a getKey method, which turns a password into a cryptographic number. Lastly, it includes a hasPassword method, which returns whether an archive is encrypted or not.

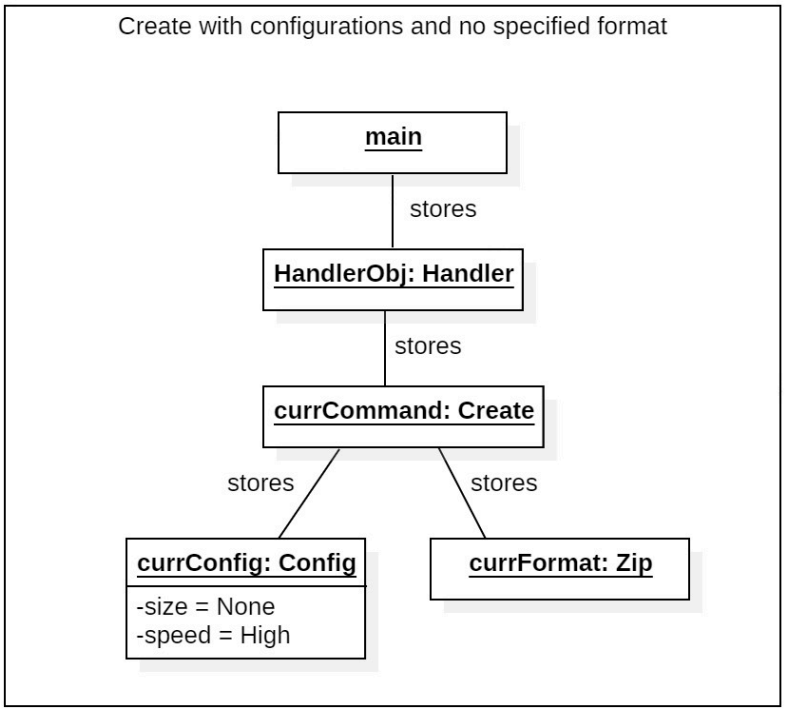
Regular classes:

- **Main:**
  This is the main class, where the main process starts. It consists of a very simple main method, which calls the loop method of a Handler instance. There is a composition between this class and the Handler class, because a Handler object cannot exist if it is not created in the Main class. The multiplicity is one to one, as only one Handler object can exist at a time.
- **Handler:**
  This class handles the interaction with the user, including user inputs. The most important method here is called loop, which provides the constant communication medium between the system and the user, the interface. The parseLine method processes the input of the user, and returns it as a triplet which consists of the command, arguments and flags of the input line. When a line is successfully parsed and a valid command has been provided, an instance of the correct subclass of Command is created inside the loop method, which allows the process to progress to executing the correct task.
- **Create:**
  This is a subclass of the Command abstract class. This class is responsible for creating an archive by implementing the run method specified in Command. It will call the compress method of the correct subclass of Format and possibly encrypt the result if the user wishes to do so. Its relationship with Command is generalisation.

- **Extract:** This is a subclass of the Command abstract class. This class is responsible for extracting an archive by implementing the run method specified in Command. It will possibly decrypt the archive had it been encrypted before and call the decompress method of the correct subclass of Format. Its relationship with Command is generalisation.
- **List:**
  This is a subclass of the Command abstract class. This class is responsible for listing the items in an archive by implementing the run method specified in Command. It will decrypt the archive and list the found items. Its relationship with Command is generalisation.
- **Config:**
  This class stores the configuration information. It has the attributes size and speed and their corresponding getters. The getters are needed in the compression process and thus are called in the compress method of the Format class. Therefore, the Config class has a directed association with the Format class. The attributes size and speed are of datatype Level and therefore the Config class has a directed association with the Level class.

Object diagram

Author(s): Yasin

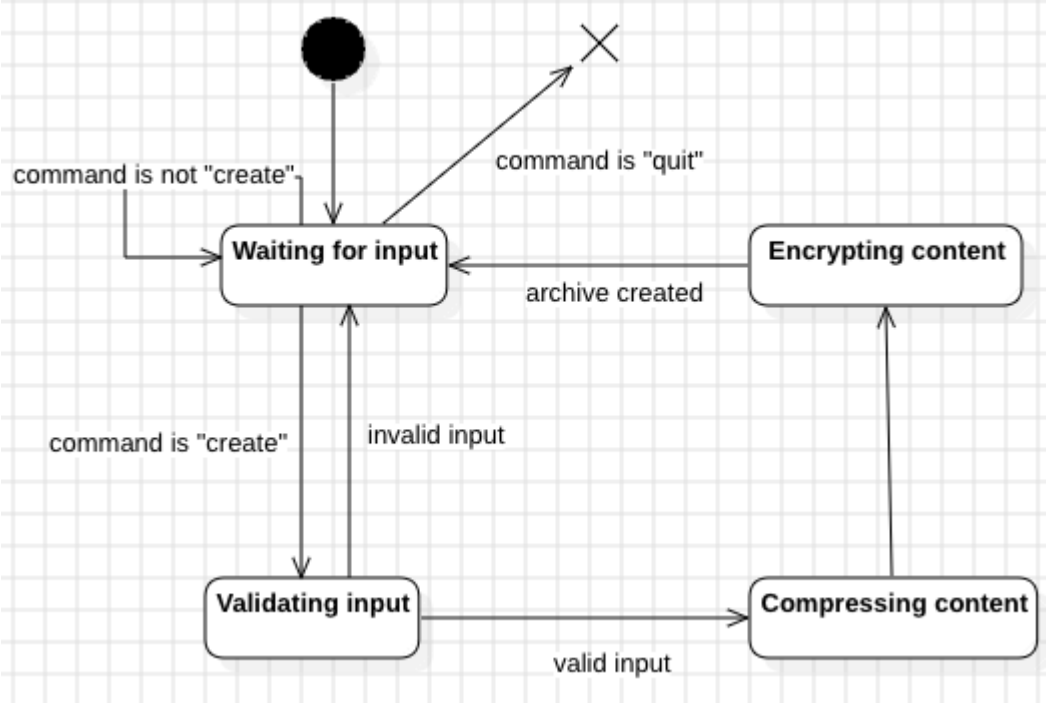


In our example above we see the scenario where a user creates a new archive. This is, in this case, done with configurations and without a specification of a format, thus the use of a our default format resulting in the creation of a Zip object. In this example, we can clearly see where the objects will be stored. If we go through the sequence, we first create a Handler object in main, we run the loop, and in the Handler object we execute one command at a time. In the Handler object we construct the Create object while giving Config and Zip objects as arguments. The create object uses the compress method in Zip which will use the getter of the Config object. This example shows how we do not store the variables in the objects themselves. The arguments are passed through the methods and thus we find no reason to store them. This is because we will only be using each variable once in a specific method and thus we find the alternative of not storing it more logical.

State machine diagrams

Author(s): Bence and Thijmen

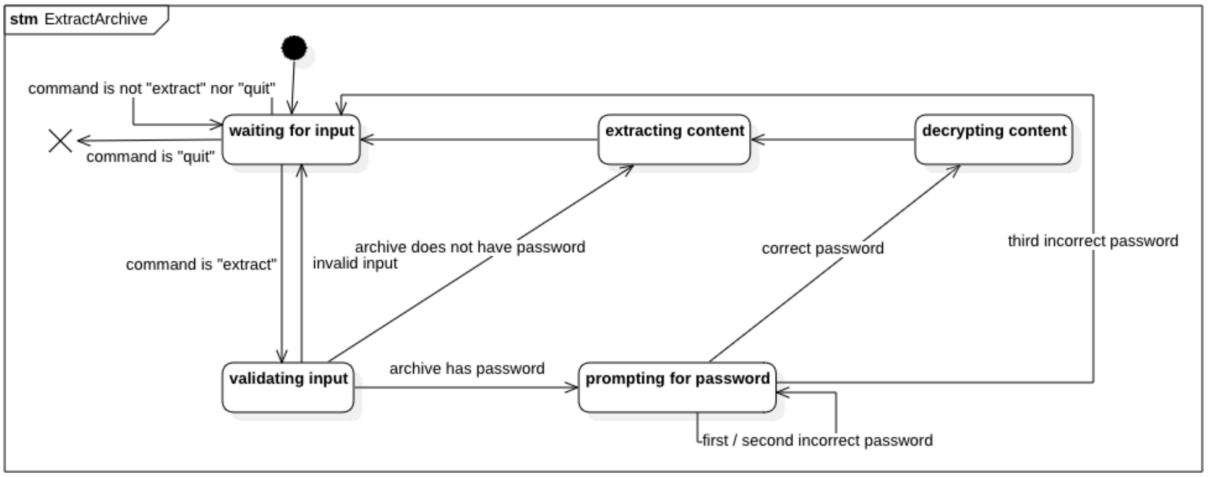
The state machine diagram below describes the process of compression.



Possible states and transitions:

- **Waiting for input:**
  During this state, the application waits for the user to provide input. If the command that the user provided is “create”, it will transition to validating the input. If the command is “quit”, the transition to the terminal state will be executed. If neither of these conditions are met, another command may be executed instead, which we are not interested in for this particular state machine diagram. Therefore, the state remains in waiting for input.
- **Validating input:**
  This state is responsible for validating the input that the user gave. If, for example, the user failed to provide the file list to be compressed, it will trigger a transition to waiting for input again. Otherwise, the process of compressing the files and creating the archive begins.
- **Compressing content:**
  In this state the process of compressing the provided files is being executed. This provides compressed content that is yet to be encrypted during the next state.
- **Encrypting content:**
  This state handles the encryption of the compressed files. After this encryption has been done, the created archive is stored in a directory and we go back to waiting for input.

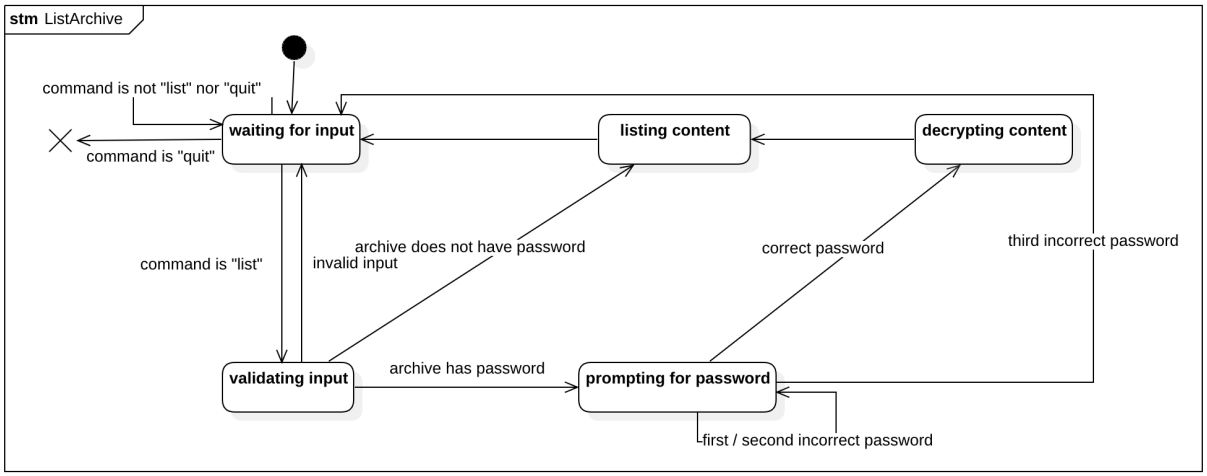
The next state machine diagram describes the process of extracting an archive.



Possible states and transitions:

- **Waiting for input**:
  During this state, the application waits for the user to provide input. If the command that the user provided is “extract”, it will transition to validating the input. If the command is “quit”, the transition to the terminal state will be executed. If neither of these conditions are met, another command may be executed instead, which we are not interested in for this particular state machine diagram. Therefore, the state remains in waiting for input.
- **Validating input**:
  This state is responsible for validating the input that the user gave. If, for example, the user failed to provide the archive to be extracted, it will trigger a transition to waiting for input again. Otherwise, if the archive to be extracted is not encrypted, the process of extracting the content begins. However, if it does have a password, a transition towards the state of prompting for the password of the archive is selected.
- **Prompting for password**:
  This state prompts the user for the password of the archive they wish to extract the content from. If they provide the correct password, the decryption of the archive begins. If the provided password is incorrect and this is the first or second guess, they will be prompted again and the state remains as is. However, if the correct password has not been provided after three guesses, the operation is cancelled and the state defaults to waiting for new input again.
- **Decrypting content**:
  This state is responsible for the decryption of the archive. After completion, it will transition to the state of extracting the content of the archive.
- **Extracting content**:
  This state handles the extraction of the content of the provided archive. Once the content has been extracted, the state transitions back to waiting for new input.

The process of executing our third and last supported command, “list”, can be found in the following diagram.



As noticeable above, there are only a few differences to be made between the state machine diagram of extracting an archive and that of listing one. These are the two differences:

- The command is “list” opposed to being “extract”.
- In the listing state machine diagram, the process prints the content of the file in “listing content”. This is different from the extracting state machine diagram, in which the content of the archive is being extracted in “extracting content”.

Sequence diagrams

Author(s): Ana-Maria

The following sequence diagram shows compression in combination with the configuration of the speed of compression.

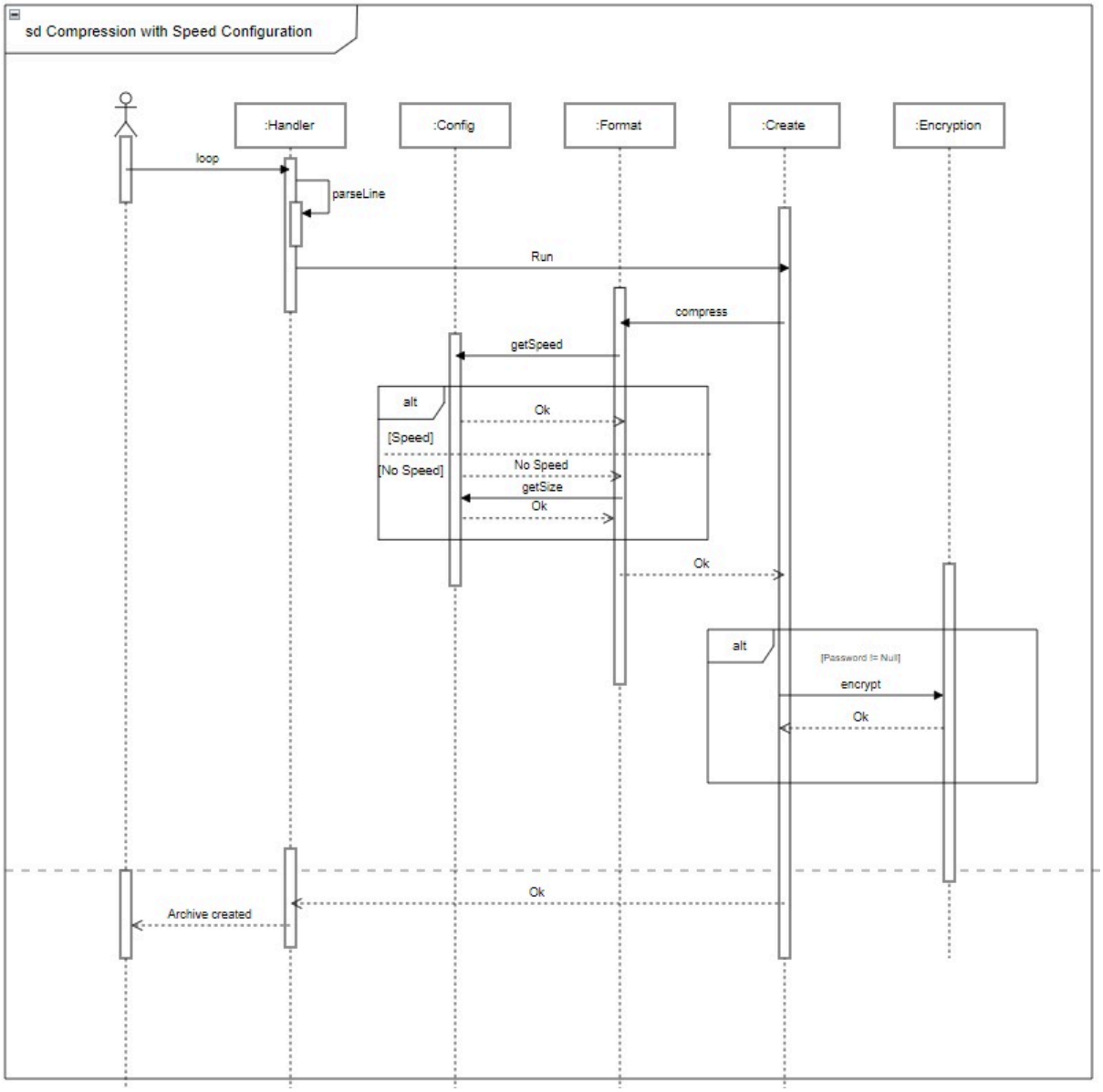


Figure 6: Sequence Diagram for Compression with Speed Configuration

In the above sequence diagram, we present the case where an archive is created with a configuration, namely the speed. The messages are all synchronous and the interaction is between instances of the classes Handler, Config, Format, Create, and Encryption. The interaction starts with the user typing a “create” command, which will be parsed in the Handler class. Then, there will be an interaction between the Handler and the Create class, which will result in the creation of a Create instance.

The Create class is called to start the compression of the files, but to do that it needs the format, so it will need to communicate with the Format instance by calling its compress() method. After which Format will interact with the Config class as it needs the configuration information for performing the Format.compress() method. Config.getSpeed() will be called and based on what it returns, there can be two cases. In order to show the two cases we created an alt combined fragment. If the user has specified in the “create” command a certain speed for the config flag, then Config will send an “Ok” response to Format to inform it that it needs to perform the compression with the speed specification and the value of the speed value. Otherwise, it will send a “No Speed” message, which will make Format ask for the size through the Config.getSize() method. Config will then respond with “Ok” and will pass the size value. These two cases are due to the specification in Feature 6 that either speed or size can be set with the -c flag. Once the compress() method is done in the Format class, it will send an “Ok” response to the Create class, indicating that it can finalise the compression process.

We decided to first compress and then encrypt the file in order to avoid as much file alteration as possible. To do this, we defined a connection between the Create and Encryption classes, meaning that after the compression is finalised in the Create class, it will not immediately inform the Handler class that the process is done. It will only encrypt in the case that a password is set which is conveyed through an alt combined fragment which displays with the guard of [Password != Null]. This is because we decided not to have a default encryption for the archives with no password. If a password was specified, then Create will ask the Encryption to perform its encrypt() method and send an “Ok” message once it is finished. Once the interaction with the Encryption is done, Create will send an “Ok” message to the Handler object to let it know that the compression is completely finalised and it can inform the user that the archive was created.

The diagram below shows the situation of extracting an archive by the use of a password.

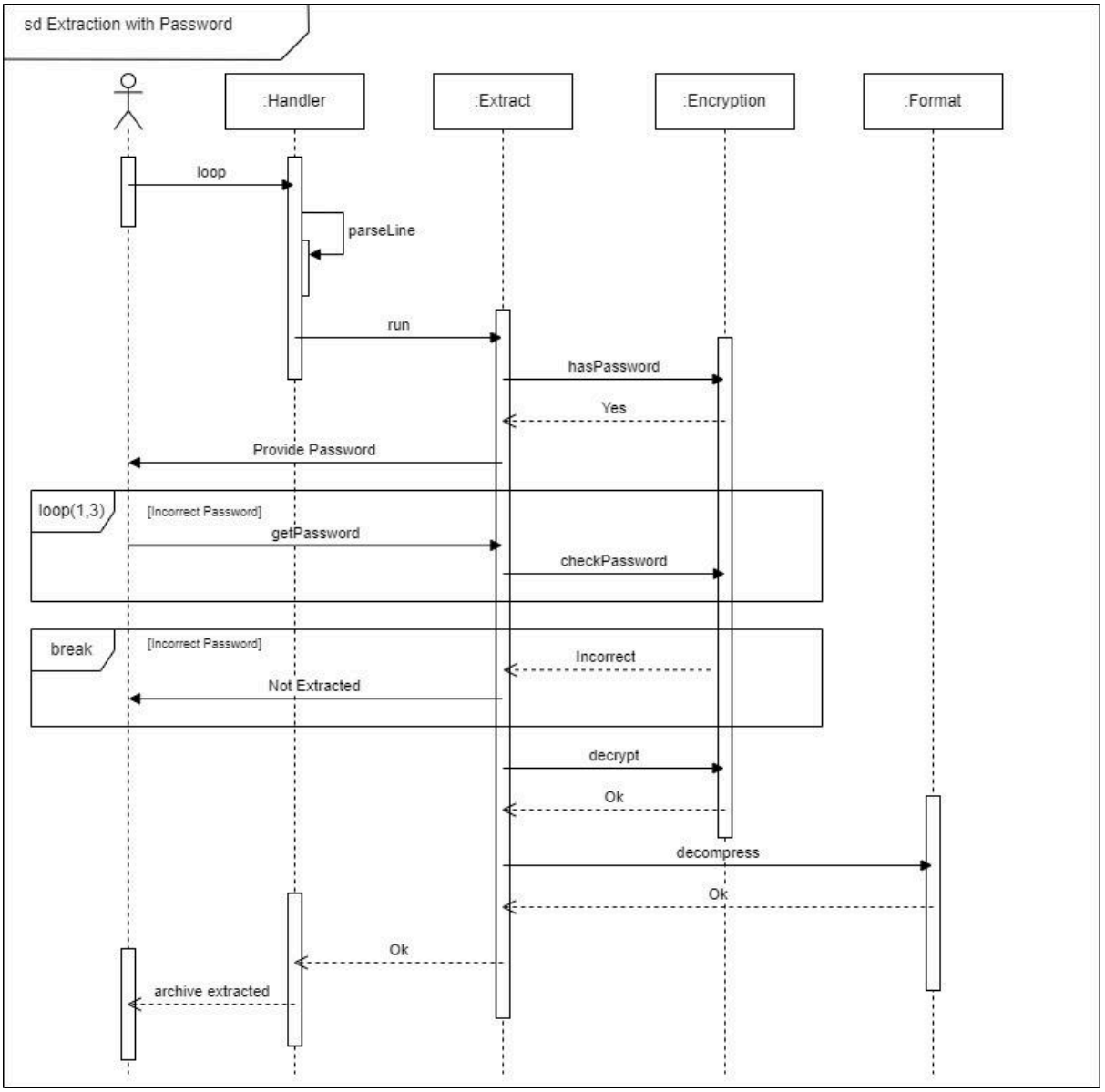


Figure 7: Sequence Diagram 2 for Extraction when there is a Password

In the sequence diagram above we present the case where a user wants to extract an archive that is encrypted. The messages are all synchronous and the interaction is between instances of the classes Handler, Extract, Encryption and Format**.** The interaction starts with the user typing an extract command, which will be parsed in the Handler class. There will then be an interaction between the Handler and the Extract class, which will result in the creation of an Extract instance. In our design, when an archive is created, the files are first compressed and then encrypted, so when an archive is extracted, it is first decrypted and then decompressed. To do this, the Extract instance will interact with the Encryption and ask through its hasPassword() method if the files were encrypted.

As we depict the case when a password was specified when the archive was created, Encryption will reply with a “Yes” message. Once the response is received, the Extract instance will directly communicate with the user by prompting them for the password. We create a loop combined fragment to indicate the fact that the user has three chances to type in the correct password. Each input will be checked in the Encryption instance by calling the

checkPassword() method. If the provided password is incorrect three times in a row, then the process will quit and the user will be prompted for a new command. So the user will have to type the extract command again in order to try to decompress the archive. Otherwise, the decrypt() method is called to decrypt the message. Once it is done, Encryption will send an “Ok” message indicating that the decompression process can be started. This is done by calling the decompress() method in the Format instance. Once the decompression is done, Format will send an “Ok” message indicating that this process is finished. After Extract receives the response, it will send an “Ok” message to the Handler instance to inform it that the archive was decrypted and decompressed. As a final step, Handler will let the user know that the extraction is successfully completed by sending the response “archive extracted”.

Time Logs

Team number	99			
Member	**Activity**	**Week number**	**Hours**	
Group	Discussed the classes to be included in the class diagram	3	2	
Group	Decided which member will do which diagram	3	1	
Bence	Worked on the Class Diagram	4	5	
Bence	Worked on State Machine Diagrams	4	1	
Thijmen	Worked on State Machine Diagrams	4	1	
Thijmen	Helped with the Class Diagram	4	2	
Yasin	Worked on the Packet Diagram	4	5	
Ana-Maria	Worked on the Sequence Diagrams	4	5	
Group	Discussed the diagrams and what should be changed	4	1	
Group	Improved the diagrams	4	3	
Group	Discussed the improved diagrams	4	2	
Bence	Wrote the explanation for the Class Diagram	4	2	
Thijmen	Wrote the explanation for the State Machine Diagrams	4	1	
Yasin	Wrote the explanation for the Packet Diagram	4	1	
Ana-Maria	Wrote the explanation for the Sequence Diagrams	4	1	
Thijmen	Produced the classes to be implemented	4	2	
		Total	35	