Assignment 3

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

IMPORTANT: In this assignment you will fully model and implement your system. The idea is that you improve your UML models by (i) applying a subset of the studied design patterns and (ii) adding any relevant implementation-specific details (e.g., classes with "technical purposes" which are not part of the domain of the system). The goal here is to improve the system in terms of maintainability, readability, evolvability, etc.

Do NOT describe the obvious in the report (e.g., we know what a `name` or `id` attribute mean), focus more on the **key design decisions** and their "**why**", the pros and cons of possible **alternative designs**, etc.

Format:
Each section header is bolded in order to differentiate between different sections. Within the text of these sections the methods are underlined, variables are *underlined and cursive*, classes are **bolded**.

Summary of changes from Assignment 2

Author(s): Yasin

Feature:

- We added an additional feature, namely analyse (description about it in assignment 1 and in class diagram)

Package diagram:

- Description on how we package
- Added and removed classes
- Added new package imports

Class diagram:

- Name changes in some classes
- Static variables and methods are now underlined
- Removed Bzip2 class
- Added Analyse class
- Removed Config class
- Added new variables and methods for Encryption
- Added new variables and methods for getting Command and Format objects
- Changed some variable and method names
- Explanation about relationships that was lacking
- Explanation for not storing arguments

State Machine Diagrams:

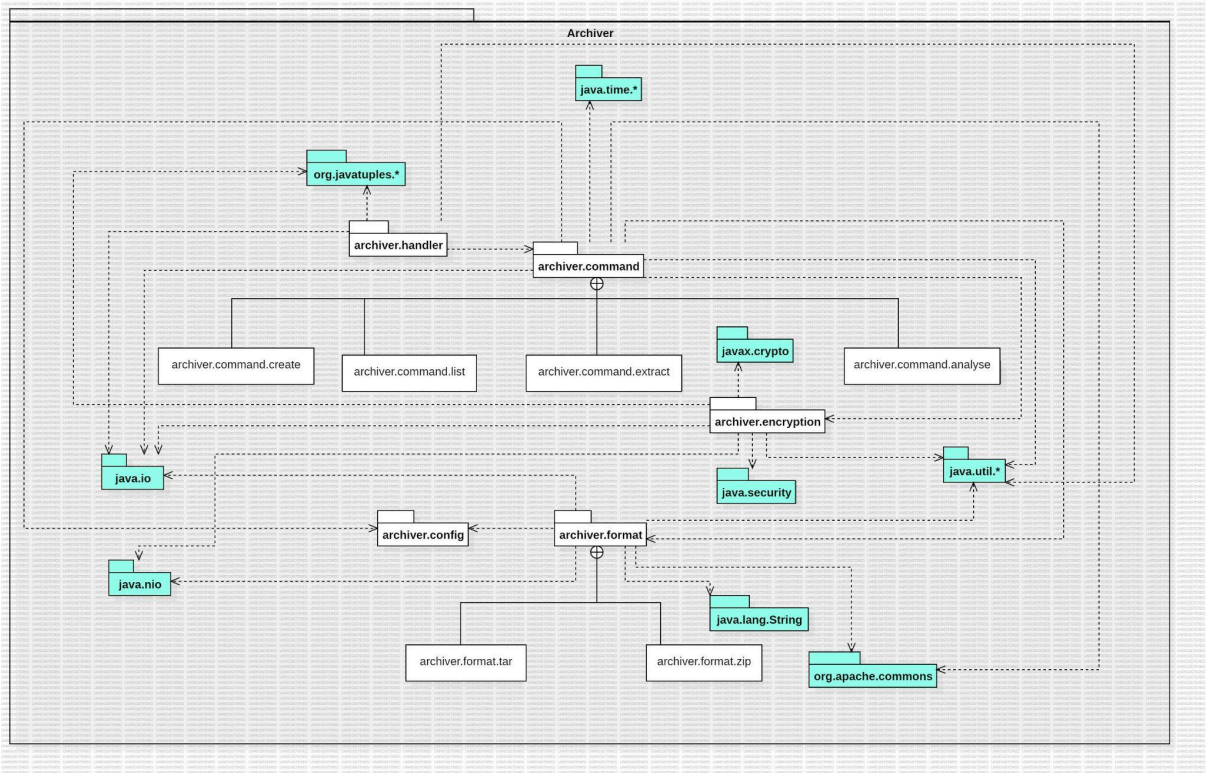
- Name changes
- New states
- Adjusted transitions
- Added guards
- Added decision nodes

Sequence Diagram:

- Added () for functions
- Omitted redundant return messages
- Added Command lifeline in both diagrams
- Generalised “Compression with speed configuration” diagram to “Compression”
- Removed Config lifeline from “Compression”
- Removed alt related to Config in “Compression”
- Changed alt related to password to opt in “Compression”
- Generalised “Extraction with Password” diagram to “Extraction
- Added new alt fragment in “Extraction”

Revised package diagram

Author(s): Yasin



The blue packages above are external packages. All dotted arrows represent an <<imports>> relationship. So for example we have that archiver.handler imports archiver.command.

Description on how we package:
We got the feedback that we should mention how we model the packages. We therefore want to mention that we model the packages by layer. This means we organise the packages according to the types of classes and the technical responsibility. This approach allows us to navigate the system easily and makes it easier to understand our system. Though the disadvantage is that it is not good at being able to isolate change. Though since we have a relatively small project, we don't suffer heavily from the disadvantage of not being good at isolating change.

Added and removed classes:
During the coding we found out that Bzip2 was not an valid format and therefore removed it from the package diagram (as well as in the class diagram). Added the **Analyse** class which is used for the bonus.

Added new package imports:
java.util
Used in order to get functionality for ArrayList, HashMap, Map and zip compression and decompression.

java.io
Used by most classes in order to make use of the Files objects, additionally it is used for input output functionality.

java.nio
Used in order to get functionality for looking for file paths.

java.time
Used by the **Analyse** class in order to get the statistic for the duration of the compression.

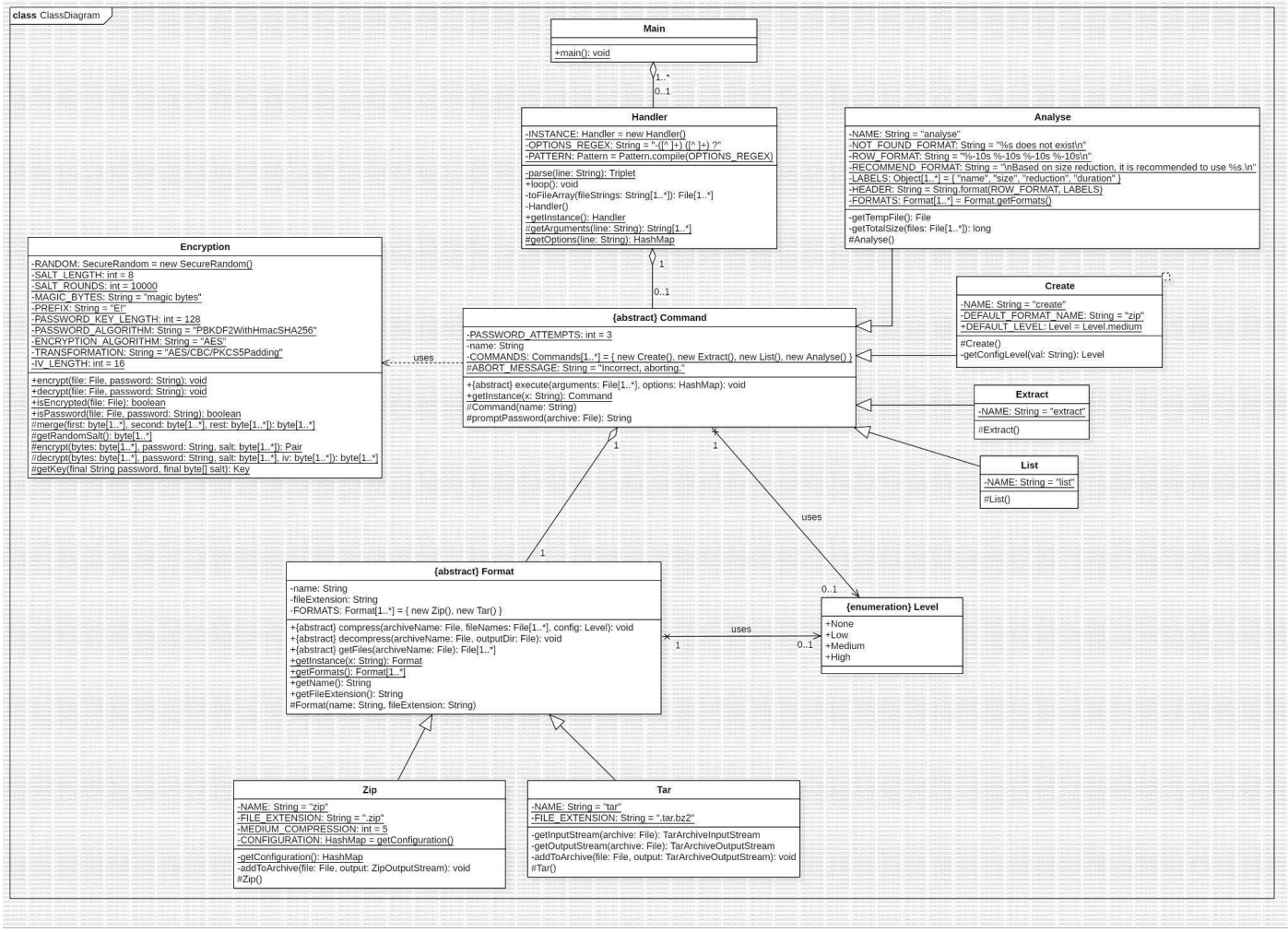
org.apache.commons
Used in order to get functionality for compressing and decompressing for the **Tar** format. Also used by the **Analyse** class in order to get access to FileUtils and Pair functionality.

org.javatuples.*
Used by **Handler** and **Encryption** in order to get access to Triplet functionality

java.security
Used in order to perform encryption and decryption functionality in **Encryption**.

Revised class diagram

Author(s): Yasin



Name changes in following classes:
Run -> execute (**Command**)
hasPassword -> isEncrypted (**Encryption**)
checkPassword -> isPassword (**Encryption**)
parseLine -> parse (**Handler**)

Static variables and methods are now underlined:
We do so in order to add clarity. We found that one of the options for displaying static methods and variables was to underline them. We ended up with a lot of static variables and methods and tried displaying them both with {static} next to the name and by underlining them. Underlining them ended up being clearer.

Removed Bzip2 class:
Since we found out that Bzip2 was not a valid format we ended up removing the class in its entirety.

Added Analyse class and prototype design pattern:

We added this class in order to implement a bonus feature. The class mostly consists of final variables used for printing the results and two methods used to gather statistics on the compression process for all formats. In *FORMATS* we make a deep copy of all the formats, this is related to Design Pattern 4. We do so with the getFormats() method in the **Format** class.

Removed Config class:
We found out that speed and level are directly related to each other and that our initial idea for the **Config** class was not going to work out. We still wanted to implement configurations for but found that we did not need a whole class for it. The “Level” enumeration ended up being enough in order to implement the config functionality that we currently required. In the **Zip** class we added the getConfiguration() method in order to get all the levels from the enum “Level” and *CONFIGURATION* variable in order to store these levels for later use when compressing. The getConfigLevel() method in the **Create** class does something similar, when given a level it returns the level associated with the string.

Added new variables and methods for Encryption:

- We initially had no clue what exactly would be needed for the encryption and therefore added a lot of variables and methods in order to implement the functionality we set out to do. To this end we want to use password-based encryption, because we want encryption keys to be based on the password provided by the user.
- These passwords need to be converted to PBEKeys which is done through the usage of a *SALT*. This salt is randomly generated. This salt needs a set amount of *SALT_ROUNDS* which increases the complexity of the key. Furthermore, this process also needs the length of the key to be derived, which is stored in *PASSWORD_KEY_LENGTH*.
- This key representing a password is then converted into a more safe key using PBKDF2, stored in *PASSWORD_ALGORITHM*. Lastly, this key is converted into a AES (*ENCRYPTION_ALGORITHM*) usable key.
- To encrypt actual bytes we specify the specific transformation to be used, which is stored in *TRANSFORMATION*. In our case, this transformation is based on AES, since that is our chosen encryption algorithm. The chosen transformation will additionally pad encrypted data.
- The *PREFIX* is stored in the beginning of the file, so we can quickly determine if a file is encrypted or not. The *MAGIC_BYTES* are used to determine whether a key is correct after decrypting.
- encrypt(bytes, password, salt) first computes the corresponding key for the password through calling getKey with the password and salt. It will then initially be a cipher corresponding to the just mentioned transformation algorithm and derived key. It will furthermore specify that the cipher is to be used for encryption. This cipher will create a unique initialization vector (iv), which is to be used to hide patterns found in the original data. To be more specific, two identical 16-bit blocks of data will not map to the same 16-bit blocks of encrypted data. This iv is derived from the cipher, because it is needed for decryption later on. The length of an iv, 16 bytes, is stored in *IV_LENGTH*. This encrypt function will return the iv and the encrypted data.

- encrypt(file, password) is similar to encrypt(bytes, password, salt), but they have been split up mainly for testing reasons. This function creates a random salt to be

used for the key. It reads the bytes of the file and then calls encrypt(bytes, password, salt) with the magic bytes + the bytes from the file as bytes. It then overwrites the original file with: prefix + salt + iv + encrypted data, where the encrypted data includes the magic bytes + the content of the file.

- decrypt(bytes, password, iv, salt) takes in a selection of bytes and its corresponding salt and iv, and a password to compute the corresponding key. It then simply decrypts the array of bytes using the derived key and iv.
- decrypt(file, password) reads the provided file. It then skips the prefix, and reads the salt and iv into a variable. Both the magic bytes, and the original content are contained in the encrypted data. This encrypted data, the password, iv, and salt are passed to decrypt(bytes, password, iv, salt) to compute the decrypted data. This data when decrypted, still contains the magic bytes in the beginning, so that has to be skipped to compute the original content of the file. The file is then overwritten with this computed original content.
- isEncrypted(file) reads the content of a file. It then simply checks if the beginning of the file matches the *PREFIX*. If this is the case, the file is said to be encrypted.
- isPassword(file, password) is used to determine if a password matches the encrypted file. It is not possible to determine the password of the file, so we simply try to decrypt it with the given password. This may throw an error, because the content might not be aligned if it is decrypted with this key. If the password is incorrect, this misalignment is often the case. However, if decryption is successful, we determine if the password is correct by validating if the decrypted magic bytes match the original magic bytes, stored in *MAGIC*.
- getKey(password, salt) creates an *ENCRYPTION_ALGORITHM* specific key from a password and salt as mentioned above.
- merge(first, second, ...rest) is a helper method which merges two or more arrays into one. This is useful for merging arrays such as the prefix + iv + encrypted data.
- getRandomSalt() generates a random salt through *RANDOM*.

Added new variables and methods for getting Command and Format objects:
We added the getInstance(x:String) method in the **Command** and **Format** classes in order to implement the Factory design pattern (design pattern 2). These methods get the corresponding command and format objects at runtime. To this end we also added the variables name and NAME in order to generate the command objects and for the format objects we have the variables *name*, *NAME*, *fileExtention* and *FILE_EXTENTION* which are used in order to search for the correct object within the getInstance methods. Lastly we have the *COMMANDS* and *FORMATS* variables used to store an array of all the implemented objects.

Added new method in order to implement singleton Handler:
In order to implement the singleton design pattern we added the public getInstance() method in the **Handler** class, additionally we set the constructor to private and add an *INSTANCE* variable. This is design pattern 1.

Explanation about relationships that was lacking:
We have received feedback that the explanation on the relationships were lacking and therefore decided to dedicate a section in order to explain it thoroughly. For each directed associate and composition we explain both sides of the relationship and the reasoning

behind their multiplicity. We start by the generalization relationships. We have that **Zip** and **Tar** extend **Format** and that **Analyze, Create, Extrac**t and **List** extend **Command**. We continue with the relationships of **Format**, we have that **Format** has a directed association with the “Level” enum, this due to the levels being used in functionality regarding compression levels, whereby we compress and decompress in the **Format** class. For each enum object we will have exactly one **Format** object associated with it. On the other side the multiplicity is either zero or one, this is due to not every single format having the functionality of compression levels, the **Tar** class for example does not have the ability to have multiple compression levels and therefore does not use the enum “Level”. In a similar fashion we have a directed association from the **Command** class to the enum “Level”. Here we again have that for each enum we have exactly one **Command** object and on the other end we have either zero or one. This is because not all commands have functionality regarding configurations, for example extract does not make use of the enum “levels”. The other relationship that **Format** has is a composition with the **Command** class. We always have that the **Command** class contains the **Format** class during the execution of a command. We always have exactly one format associated with a command since the file archiver does not allow for a file to be associated with multiple formats. And we have exactly one command associated with one format object since only one command can be executed at a time and therefore we only have at most one command in the system. Lastly this is a containment because the way we get the **Format** object is by passing a reference to an already existing **Format** object with the method getInstance(x), so therefore when a **Command** object is terminated we still have that the **Format** objects keep existing. The **Command** class depends on the **Encryption** in order to implement encryption and decryption functionality. We now look at the composition relationship between the **Command** class and the **Handler** class. We decided for a containment because the way we get the **Command** object is by passing a reference to an already existing **Command** object with the method getInstance(x), so therefore when the **Handler** object is terminated we still have that the **Command** objects keep existing. The chosen multiplicity is decided as follows, for each **Command** object there has to be exactly one **Handler** object, this is because the command is given and executed by the handler, and since we implement a singleton handler we only have one handler in the system and therefore there can only be at most and at least one **Handler** object for each **Command** object. For each **Handler** object we either have zero or one **Command** object. During execution we handle one command at a time so there can be at most one **Command** object per **Handler** object, at the same time we find that we don’t at all time execute a command and in such times the amount of **Command** objects contained in the **Handler** object is zero. Lastly, the relationship between **Main** and **Handler**. It is a composition since the **Handler** object won’t terminate in the case that a **Main** object terminates. We have that for each **Handler** object there has to be at least one **Main** object but there is no upper limit, this is due to the singleton design pattern implementation, we therefore have that all the users have access to the same object. On the other end we have that for each **Main** object we have in terms of multiplicity either zero (they don’t use the system) or one (they use the system).

Explanation for not storing arguments:
We initially decided against storing the arguments of the commands as variables and received the feedback that we lack this explanation. We decided not to store them since we did not implement undo and therefore never use the arguments of the commands after the execution of the execute method.

Application of design patterns

Author(s): Ana-Maria Musca

	DP1
Design pattern	Singleton
Problem	The **Handler** class is the base of our archiver. In the **Handler** class we handle the user input and output. This means that we parse the input in a specific pattern, we instantiate (and execute) the indicated command object and we print messages in order to relay information in terms of validation or task completion. It could be the case that someone modifies variables and objects within the **Handler** object and thus two users in the same system might have different behaviours due to using different instances of the **Handler** object. When a change occurs we want this change to be reflected in all instances of the **Handler** object.
Solution	We incorporate the singleton design pattern in our implementation. We do so by creating a single instance of the **Handler** class and provide global access to it. So when the users ask to get an instance of the **Handler** class the getInstance method returns the same **Handler** object. Therefore any changes made to the **Handler** object will occur throughout the whole system.
Intended use	When the application is started, we instantiate a **Handler** object. When a user asks for an instance of a handler object it will need to call getInstance. by calling getInstance we return an instance of the **Handler** class that was created at the start of the program, instead of creating a new one. After this inputStreams are received from the command line, parsed and and then **Command** objects are instantiated.
Constraints	None.
Additional remarks	None.

	DP2
Design pattern	Factory Method
Problem	We support a multitude of commands and formats in our file archiver. We have noticed that this adds dependency between different classes. When making a change in the various supported commands we had to make changes in the **Handler** class. The same issue occurred when we wanted to extend the amount of formats, we had to make changes in the commands of the **Command** class. We want to avoid the amount of places where we need to modify code in order to have ease of extendability.
Solution	To this end we implemented a form of the Factory method design pattern.

	We did so for the **Command** class and the **Format** class. By doing so we only need to make changes in the newly implemented subclasses and their corresponding superclass. This allows ease of extendability of the system and reduces the amount of dependencies.
Intended use	**For the Factory method in Command:** When an inputStream is received in the **Handler** class, it is parsed and a Command object has to be instantiated, so Command.getInstance(type) will be called. If the type is not an implemented one (in our case create, extract, list or archive) the method will return a null object and the user will receive a “command not found” message. If the type is implemented then **Command** will return a reference to an already-existing object of the specified type and the process of performing that command is continued. **For the Factory method in Format:** When Create.execute is called the **Command** object needs a **Format** object in order to perform format specific functionality, so Format.getInstance(type) will be called. If the compression format is not implemented (in our case zip and tar) by our archiver, then a null object will be returned and the user will receive a “Compression format does not exist.” message. If the type is implemented then it will return a reference to an already-specified object of that type.
Constraints	None
Additional remarks	In the case that you want to change the command name of a command you will have to do so in the subclass of the command.

	DP3
Design pattern	Command
Problem	Our archiver needs to include four types of commands, namely create, extract, list and analyse. Each one of them has a different implementation and purpose, but at the core, they are types of commands.
Solution	We integrate the Command design pattern in our file archiver by implementing an abstract **Command** class with an abstract method called execute. Each command (create, extract, list and analyse) will correspond with a class which extends the abstract class and implements in its own way the execute method. The **Command** class acts, also, as an intermediate between its subclasses and the **Handler** class (we can also consider them as two different layers: input layer and execute layer), as only after the command line input is parsed a **Command** subclass is instantiating and the command is executed. In this way we separate the parsing of the command line interface input from the actual execution of the requested command.
Intended use	When implementing new commands all you need to do is extend the **Command** class and implement the execute method. Additionally you would need to set the name of the command and add the command to the

	COMMANDS array stored in the **Command** class. The system will get the corresponding **Command** object by using the factory method implemented in design and afterwards will call Command.execute in order to execute the functionality of the corresponding command.
Constraints	None.
Additional remarks	None.

	DP4
Design pattern	Prototype
Problem	When an analyse command is performed, analyse needs to know which formats are supported by the archiver. Afterwards it will iterate over them and get the statistics regarding the compression process for each format. As a result, we need to find a way to expose the formats that extend the **Format** class, but want to do so without allowing direct modifications to the objects. Since modification to these objects could affect different parts of the system.
Solution	We implement the prototype design pattern in the **Format** class in order to create a deep copy of each of the implemented formats, in order to ensure that their internal structure cannot be modified.
Intended use	When you want deep copies of all the implemented formats, which is currently only the case in the **Analyse** class, you call the Formats.getFormats method. This method returns an array of deep copies of all the implemented formats.
Constraints	None.
Additional remarks	None.

Revised state machine diagrams

Author(s): Bence and Thijmen

Overall improvements over all state machine diagrams

Name changes:

The UML theory states that the name of any state is required to be a noun. Therefore in the revised version we ended up changing all the names of our previous states to align with the theory. Each section we will start with the name changes that have been made in the state diagram. Additionally the name changes sometimes lead to a splitting one state into several, when this happens we will mention what every single individual state now represents in the *"Description changes and additions"* section.

New states:

During the coding we encountered some issues and after solving the issues we ended up needing some additional states (and their corresponding transitions). We will list all the new states in the *"New states"* section of the diagrams and their description in the *"Description changes and additions"* section.

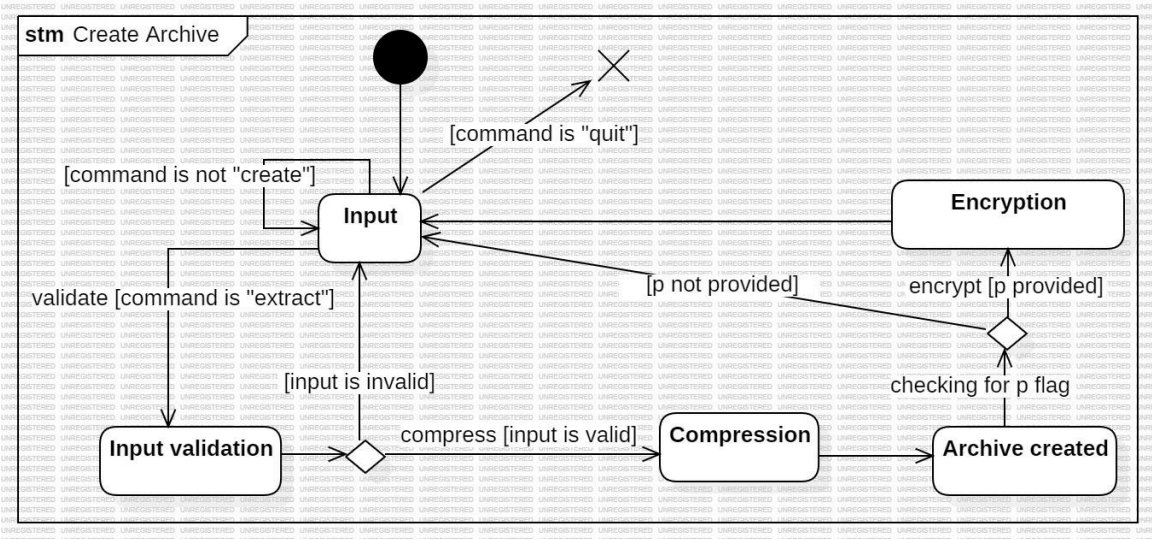
Transition changes:

Due to us mixing up verbs and nouns for states in the first version we improperly modelled the transitions. We make up for this now by clearly stating what is being performed and the guard required for this transition. We derived these transitions by transforming the state into a noun and adding the missing verb part to the transition.
So for example "Compressing content" is now the state "Compression" and the transition that enters the state is "compress[valid input]"

Added guards and decision nodes:

Which brings us to the last overall improvement. We added guards where we thought it to be appropriate. This way we clearly convey the idea of something being checked. Initially we implied there being guards by naming the transitions specific events. But now we explicitly mention the guards, which also allows us to show what transition is occurring. With the addition of guards we also thought of the addition of decision nodes since they would allow us to clearly convey the idea that a decision is being made, thus we also added those in spots we found important to show a branching.

The state machine diagram below describes the process of creating an archive.



The following name changes have been made:
Waiting for input ->Input
Validating input -> Input validation
Compressing content -> Compression
Encrypting content -> Encryption

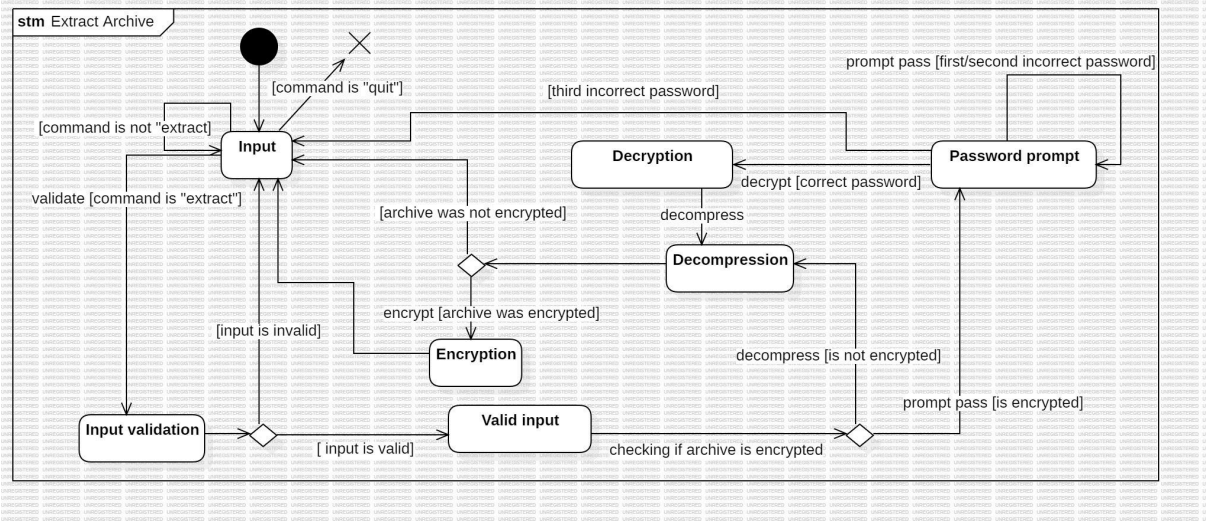
New states:

- **Archive created:**

Description changes and additions:

- **Archive created:**
  Right after we are done compressing we want to check if the file will be encrypted. This state represents the state the program is in between these two transitions, it's a temporary state that conveys the idea that the archive is now completely created.

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



The following name changes have been made:

- Waiting for input ->Input
- Validating input -> Input validation & Valid input (split it up into two)
- Prompting for password -> Password prompt
- Decrypting content -> Decryption
- Extracting content -> Decompression

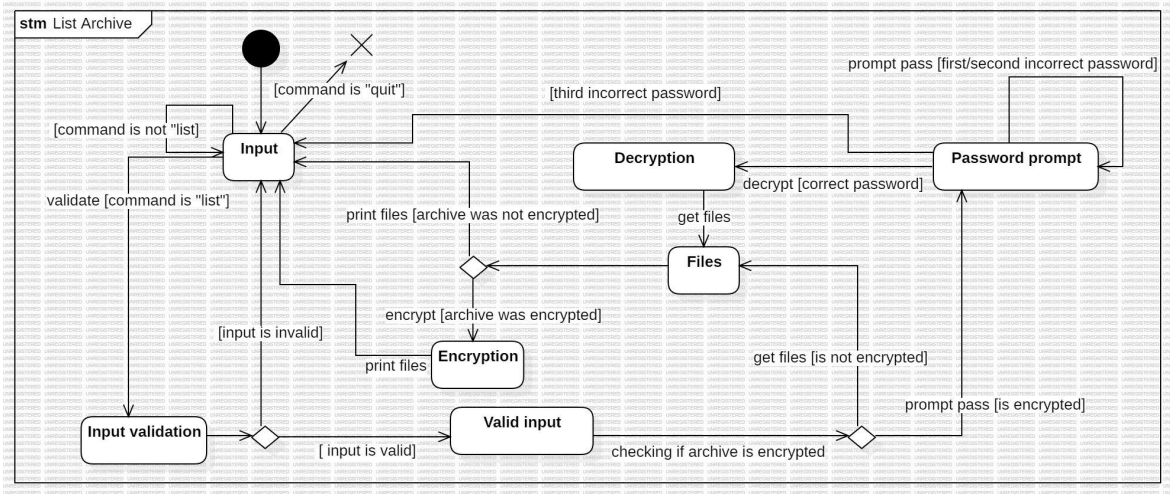
New states:

- Valid input
- Encryption

Description changes and additions:

- **Input validation**:
  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.
- **Valid Input:**
  This state represents the state we are in after we confirm that the input is valid. We now need to check if the file is encrypted. 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.
- **Decompression:**
  This state handles the decompression of the content of the provided archive. Once the content has been decompressed, the state either transitions back to waiting for new input or it transitions into the state “Encryption”.
- **Encryption:**
  This state handles the encryption of the decompressed file. After this encryption has been done we go back to waiting for input. This state is of importance since if we were to exclude this then the file we just extracted would no longer be encrypted (since during the extraction process we decrypted it).

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



We still have that the two state diagrams for listing and extracting an archive are similar and thus the changes made for the “Extract Archive” state machine diagram are identical to the changes made in the “List Archive” state machine diagram.

The following name changes have been made:

- Listing content -> Files

Description changes and additions:

- **Files:**
  After we have confirmed that the input is valid (or after decrypting for encrypted archives) we get the files that reside in the archive, we do so in order to print the file names and their corresponding path at the end of the process. In this state the files have been received.
- **Encryption:**
  This state handles the encryption of the decompressed file. After this encryption has been done we print the files names and then go back to waiting for input. This state is of importance since if we were to exclude this then the file we just extracted would no longer be encrypted (since during the extraction process we decrypted it).

Revised sequence diagrams

Author(s): Yasin

Overall changes:

For assignment 2 we received the feedback that we should add parentheses for methods. Therefore in order to clarify which methods are called (and with which arguments), we added the parentheses and the arguments used when the method is called. Additionally we slightly changed the names of the messages in order to align with the method names used in the implementation. Furthermore we chose to exclude some return messages for which it was clear what is being returned, according to the theory this is something you are allowed to do and we chose to do so since we found that it added clarity and space. Moreover, due to implementing the Factory Method design pattern we now have an additional “Command” lifeline in both diagrams and additional getInstance() methods which are used in order to get a **Command** or **Format** object. Additionally, due to changing the **Handler** class to a singleton we have an additional getInstance() method in order to get the singleton object. Lastly, we added a “prompt input” message at the end of both sequence diagrams in order to convey the idea that we have finished the command and now we start again, this gives better closure to the whole operation.

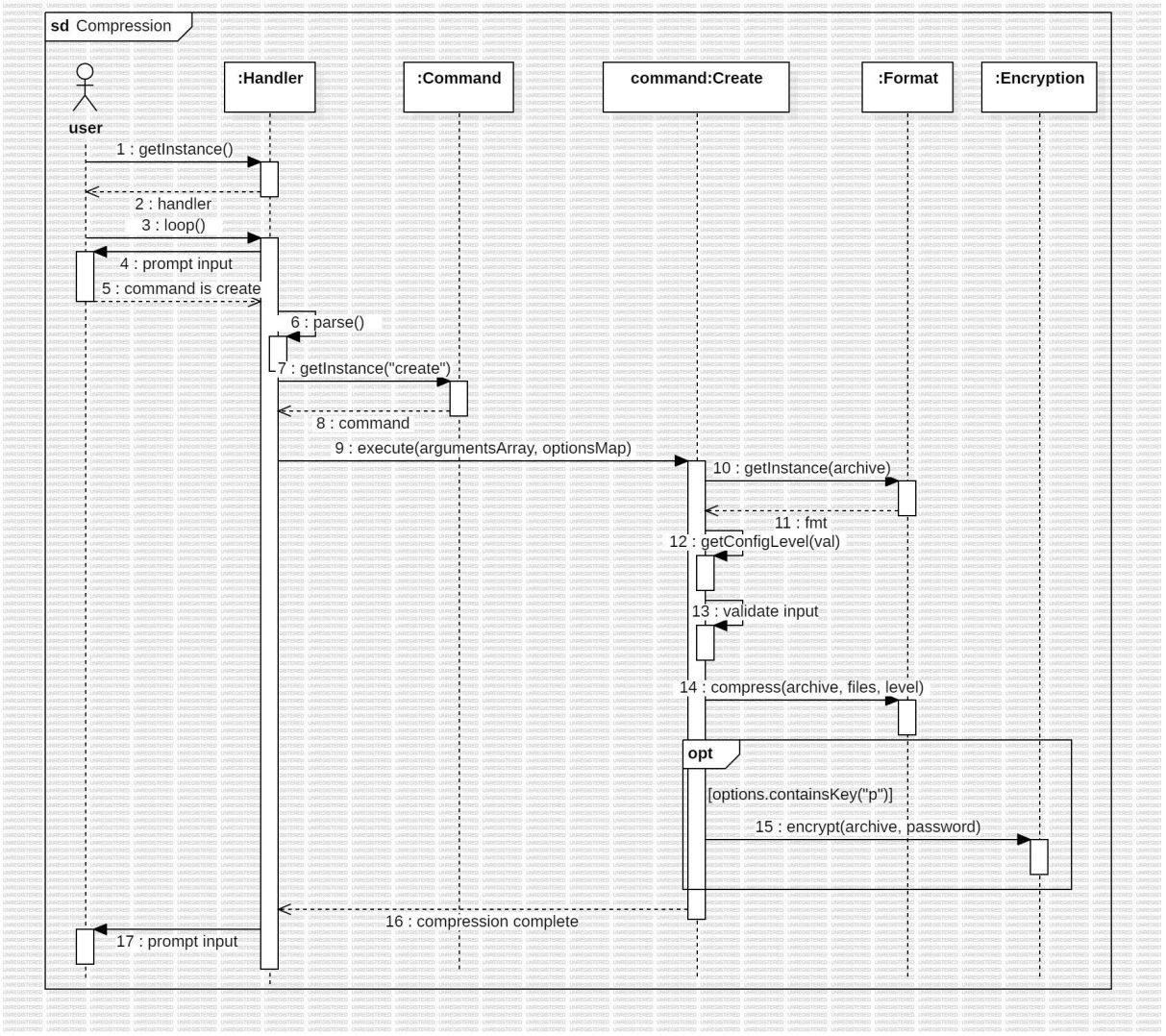


Figure 6: Sequence Diagram for Compression

Generalised “Compression with speed configuration” diagram to “Compression”:
For our initial sequence diagram we wanted to show compression with the speed configuration. We no longer implement config in the same way, now you can only choose the compression level. So it is no longer a choice between either compression speed or compression level, whereby you could have specified either. Therefore we chose to omit this detail of “with speed configuration” and decided to show all the interactions that occur when a create command is executed and so we generalised the sequence diagram to “Compression”.

Removed Config lifeline from “Compression”:
Like we previously mentioned, we implement “Config” in a different way than we initially thought and therefore ended up removing the “Config” lifeline. Additionally we have removed the alt related to the “Config” lifeline. This interaction of getting the config is now done in one message, namely message 11 : getConfigLevel(val).

Changed alt related to password to opt in “Compression”:
Initially we believed it appropriate to model this operation with an alt fragment, but we found that an alt fragment does not allow one operand and therefore can not be used to represent an if statement. Instead the opt fragment is the one that represents an if statement. So, in order to add accuracy, we changed the alt fragment into an opt.

Generalised “Extraction with Password” diagram to “Decompression”:
For our initial sequence diagram we wanted to show the flow of interactions that occurred when someone wanted to extract an archive that was encrypted with a password. We succeeded but found that when making such a specification it’s misleading and misses an important part of the interaction. Therefore we decided to generalise the sequence diagram by considering decompression both for an archive that has been encrypted and for one that has not been encrypted. In order to do so we added an additional alt fragment and put the loop and the break fragments in one of the operands of the alt fragment. When the break guard is true we go back to the handler and leave the alt fragment in order to continue asking the user for a new command.

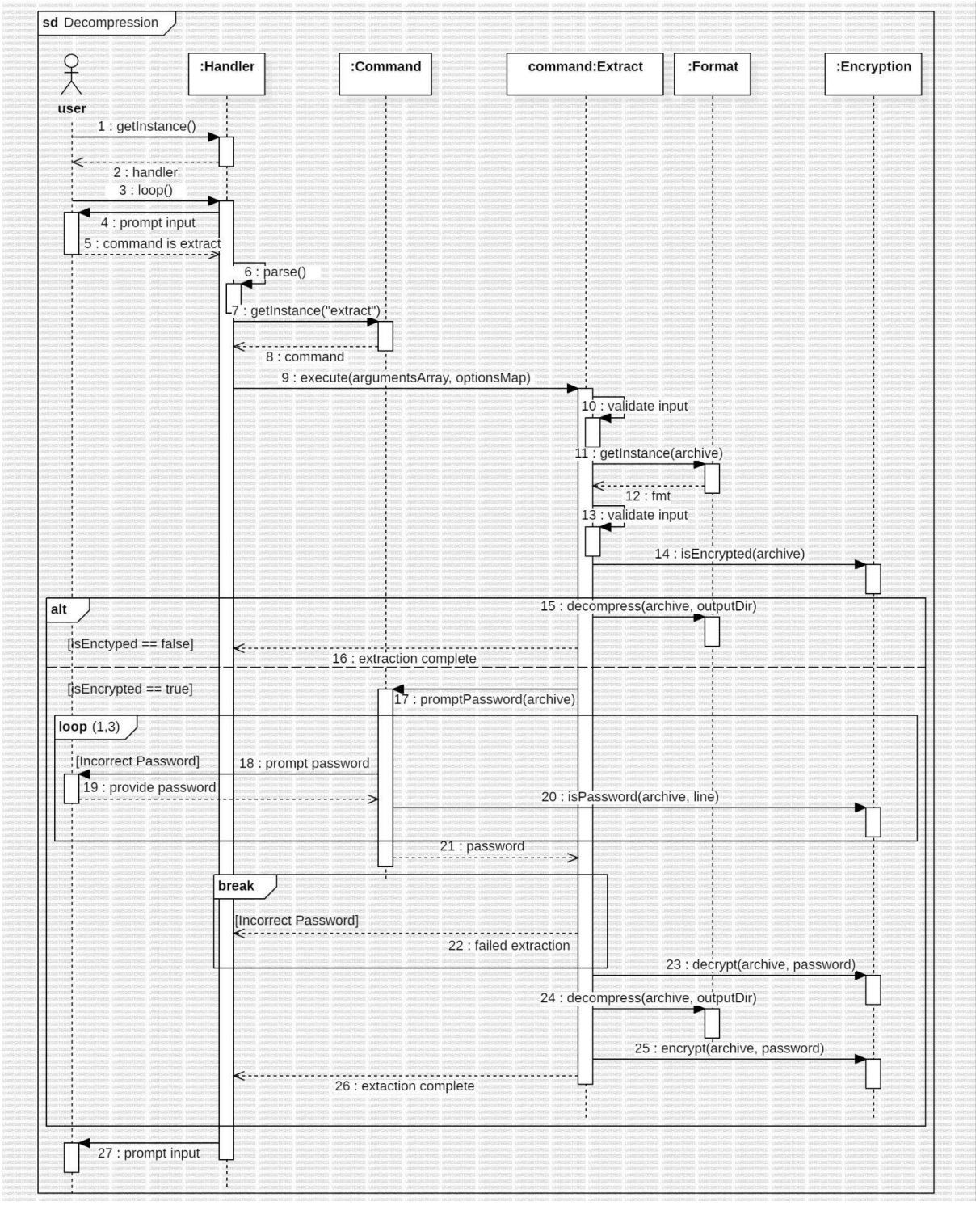


Figure 7: Sequence Diagram for Decompression

Implementation

Author: Bence Matajsz

Strategy:

First, we decided to split the work by features, each person being responsible for the features they were assigned during assignment one. This left each person of the team implementing their assigned features on their own. To make sure that all operations run smoothly and we do not have any misunderstanding, we would regularly check up on each other during the implementation of each feature, and discuss the decisions being made regarding the design of the current feature. In case the responsible person ran into any problems, other members of the team provided help. We would have regular meetings to discuss current progress, alternative design ideas, and what is next. We implemented the features in a systematic way, increasing order based on the number of the current feature, however, if an implementation took relatively too much time, we already started making drafts of the next task. We used Git and GitHub as version control tools throughout the whole project. The general principles were the following:

- Create a new branch for every assignment, feature, task, fixes, etc.
- Create frequent commits during, and at the end of each coding session.
- Create pull requests before merging branches. This way other members of the team can see the changes, and either approve them or request changes.
- Create issues and assign them to the responsible person to easily keep track of what is left to do.

Solutions:

Structure setup:

In order to start implementing feature one, we first needed to lay down the base and implement the general functionalities of the program. Create the Main and Handler classes, the abstract classes for formats and commands, and the Create and Zip class for the first feature to use. The Main class utilises the singleton design pattern with Handler class, by creating exactly one instance of this class. The loop method of this Handler object controls the flow of the whole program from this point. The Handler class also handles parsing user input. The input line is processed using regular expressions and the command, arguments and flags are stored for later use.

Feature 1:

The first step to implementing the create functionality is retrieving the Create object using the factory method, if the user-given command is “create”. Once we have this object, its execute method is called, which processes the passed arguments and flags, and checks if they are valid (number of arguments, format, etc.). If everything is valid, the object of the specified or default format is retrieved using Format’s factory method implementation, and the compress method of this object is called, which compresses the given files and creates the archive.

Feature 2:

To implement the extract feature we could already build on the base we created during the

implementation of feature one. To make “extract” a valid command, we had to add it to the possible commands in the Command class. After implementing input and argument validation for this feature (checking if archive exists, format, etc.) by overriding the execute method of Command in the Extract class, the decompression algorithm was implemented for the default zip format in the decompress method of the Zip class.

Feature 3:
The encryption feature is done in the Encryption class, which offers fully static functionality in terms of methods and attributes. The logic and implementation of both the encryption and decryption can be found here, along with some helper functions to support the operations. The “-p” flag has been added to the possible flags when archive creation is initiated, and code has been added to the Create class that calls the encrypt method of Encryption, encrypting the archive with a user-given password, if this flag is found. Also, to make decryption of an encrypted archive possible, the user is prompted to enter a password in the execute method of the Extract class, if the given archive happens to be encrypted, which is checked by the isEncrypted method of the Encryption class. If this is the case, the archive gets decrypted then decompressed to create the output directory, then re-encrypted to restore the archive.

Feature 4:
The list feature is very straightforward. The execute method of the List class does some argument validation and error checking similar to extract, including checking if the archive exists or its format is supported. After that, similarly to the extract feature it is checked if the archive is encrypted. If yes, it is decrypted then the files are listed and the archive gets re-encrypted, if not, the files simply get listed.

Feature 5:
For feature 5 we have decided to add support for tar format. There were multiple choices for the compression algorithm to use for this feature, such as GZIP or BZIP2. We decided to go with the latter, and integrate all the previous features with it (compress, decompress, list). This feature is implemented in a way that it allows new formats to be added to the system and offer integration with the functionalities.

Feature 6:
The last basic feature is configuration, which has ended up going down a different route than what we first specified in assignment one. Since the level of compression and speed of compression are directly related to each other, we decided to only include the level of compression in our implementation. We further decided to remove the Config class as it did not make any sense to keep. It did not add any extra functionality to the program, the Level enum already had all the possible levels defined. This feature adds sense to the “-c” flag, in the Create class the given arguments for the configuration are generalised and passed to the object of the correct subclass of Format, where the given, generalised compression level is translated to the format-specific compression level configuration. If the format does not allow configuration, the user-given configuration is ignored and the default native algorithm is used.

Feature 7:
The bonus feature implements analysing compression and printing out statistics. This is

done in the Analyse class which extends Command, and the main functionality is implemented in the execute method. This functionality outputs and measures the size, reduction percentage, and speed of each supported compression format using the default settings. This is done by comparing these statistics before and after compression, and calculating the results after.

Design patterns:
In overall, we managed to stick to the core design of our system specified in the previous assignments and UML diagrams. Most of the challenging decisions we had to make were regarding the design patterns to implement. Although our initial implementation already had similarities to actual design patterns, at the end we had to decide which actual design patterns we should tailor our implementations to. We already had a singleton design pattern with the Handler class, and we figured that we could implement the template design pattern with the Command class, however we decided not to, because we thought there is not enough overlap between the commands. Instead, we implemented the command design pattern, defining the interface for all commands. We also thought about implementing undo/redo functionality since we now have the command design pattern implemented, however we ended up ruling the idea out because it seemed unnecessary. Furthermore, the factory method is also used in Command, at startup an instance of each Command subclass is created, and they can be retrieved using the getInstance method of the Command class. We also thought about adding a dedicated Factory class, but decided not to as it would not add much functionality. The factory method design pattern is further used with the Format abstract class, which builds on the same principles.

Path to Main class: /src/main/java/archiver/Main.java
Path to the JAR file: /build/libs/software-design-vu-2024-1.0-SNAPSHOT.jar
Link to demo video: https://youtu.be/MWkhbOG43_Q

Maximum number of pages for this section: 4

Time logs

Time log A3

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