Assignment 3

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:

- **Class** in bold
- *Functions/Methods* in cursive
- Interfaces underlined

Summary of changes from Assignment 2

Author(s): Joli-Coeur Weibolt

Package Diagram

Based off the feedback on our previous assignment we changed our package diagram in a way that shows the import each package has. We also described our approach when making the package diagram.

Class Diagram

Based off the feedback on our previous assignment we changed the following:

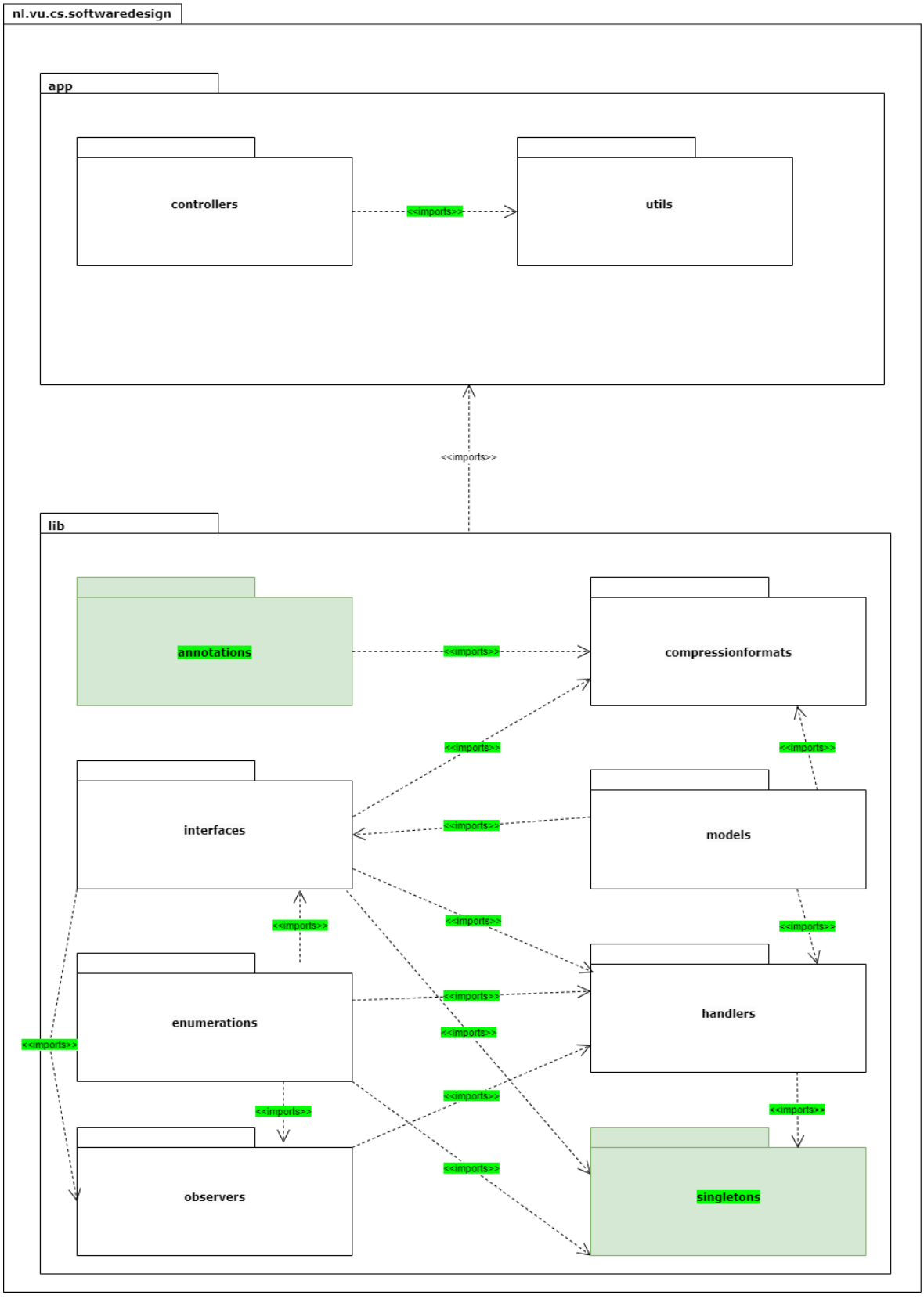
- We added the missing multiplicities, on the associations.
- We added a few handlers and classes, because we use them in assignment 3 and therefore need to add them is assignment 2.
- We replaced a few classes. For example, we replaced the **contentsInserter** and **contentExtractor** with an **ArchiveHandler**.
- We removed the Folder class, because we actually didn't need to make this class anymore.
- The rest of the changes where very small, we changed the data type of some functions.

State Diagram

Based off the feedback on our previous assignment we made sure that the names of the states are nouns, which indicate that an object is in a state. Aside from the names of the states we made sure to implement guard conditions. The feedback stated that our guard conditions should be events, so we changed our guard conditions to events. Lastly, we made sure to use the decision node properly.

Revised package diagram

Author(s): Jaïr Telting, Dennis Moes



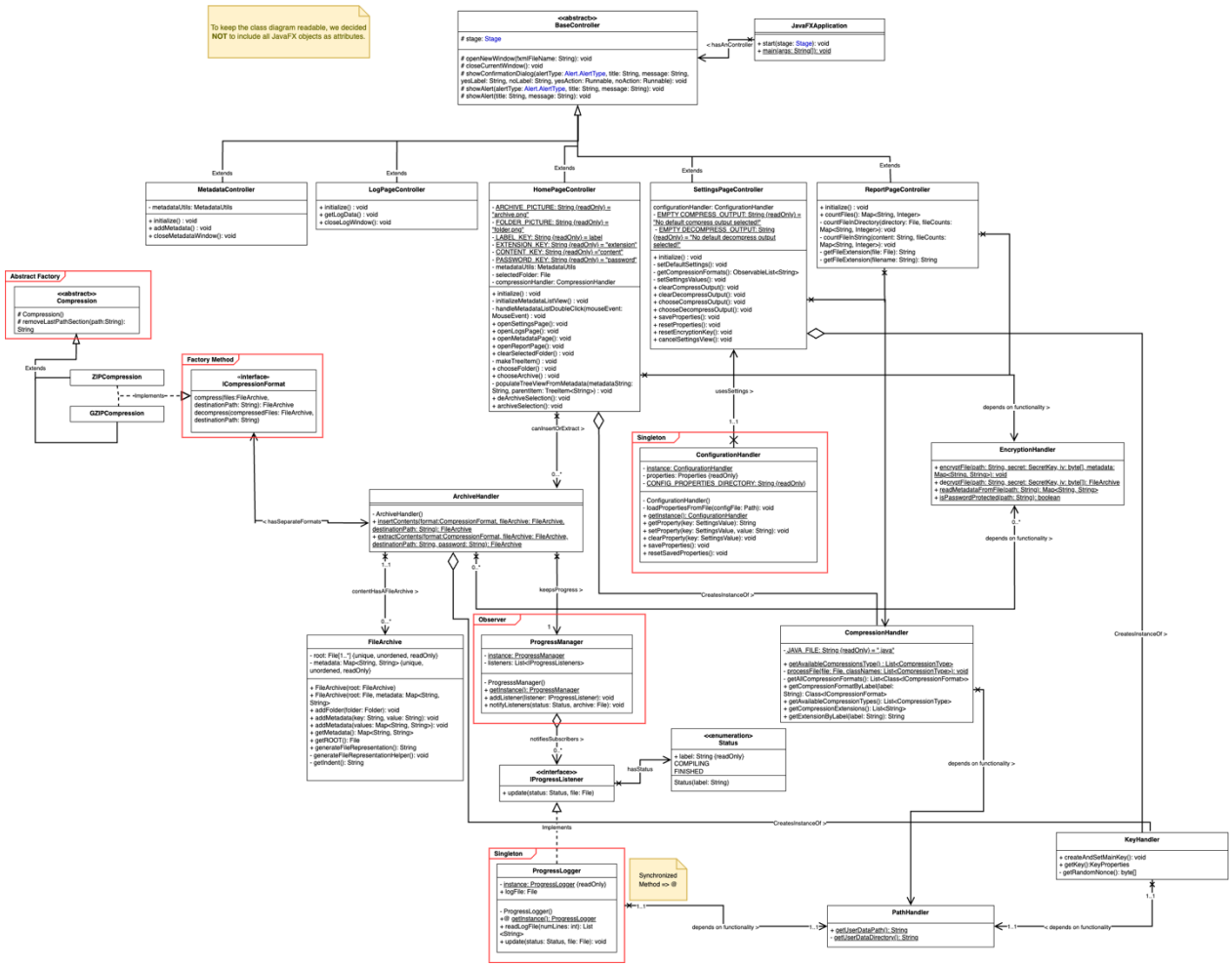
Several changes were made on the Package diagram. Firstly, all additional packages were added to the package diagram. Secondly, we have corrected and included all the <<*imports*>> relationships present in our project. Thirdly, we changed the naming of the packages to conform to the java naming conventions.

We opted to package by layer. Each layer (package) focuses on a specific aspect of the system, such as presentation. The package includes the packages:

- nl.vu.cs.softwaredesign.app -> Is the main application package
    - nl.vu.cs.softwaredesign.app.controllers -> The controllers responsible for handling user request
    - nl.vu.cs.softwaredesign.app.utils -> Contains utility classes or helper functions used across different parts of the application.
- nl.vu.cs.softwaredesign.lib - > Library package containing different types of utility classes, interfaces, models, enumerations, handlers, observers, and singletons.

Revised class diagram

Author(s): Simon Vriesema
You can find the class diagram: [here]



Singleton classes

ProgressLogger

Manages logging progress updates into a single log file. Implemented as a singleton to ensure single instance and avoid issues with multiple log file accesses.

ConfigurationHandler

Maintains a single configuration file for consistent settings. Loads configurations into a local variable, allowing read and write operations on it.

Interfaces

ICompressionFormat

Provides a standardized framework for integrating compression formats seamlessly. New formats can be added by implementing this interface.

IProgressListener

Defines essential functionality for listeners within the observer pattern, focusing on progress updates.

Observers

ProgressManager

Notifies subscribers (**IProgressListener** implementations) about changes in progress status, enabling automatic logging and other relevant actions.

Enumerations

Status

Conveys the current state of the archive within the application, facilitating accurate recording of relevant status updates.

Abstract classes

Compression

Provides common compression operations, ensuring efficient path manipulation within compression formats.

Regular classes

ArchiveHandler

Manages file archives efficiently, handling insertion and extraction operations seamlessly.

FileArchive

Organizes and manages file archives, systematically tracking and categorizing files into folders.

JavaFX controllers

BaseController

Abstract class shared by all controllers, containing important shared code and functions for navigation and alerts.

HomePageController

Manages interactions for the application's home page, handling folder and archive selection, compression, decompression, metadata management, and navigation.

SettingsPageController

Manages the settings page, allowing configuration of compression formats and default output directories, along with saving, resetting, and encryption key management.

ReportPageController

Manages the report page, displaying file counts based on selected files or archives in a bar chart.

MetadataController

Manages the metadata window, allowing users to add, edit, and close metadata key-value pairs.

LogPageController

Manages the log page window, displaying log data from the ProgressLogger singleton instance and providing functionality to close the window.

Improvements

ArchiveHandler

Consolidated insertion and extraction operations into a single class for streamlined codebase and unified interface.

Compression

Introduced **AbstractCompression** class for common functionalities across compression formats, promoting code reusability and consistency.

ConfigurationHandler

Enhanced functionality to efficiently load, clear, and reset properties, improving management of configuration properties.

Application of design patterns

Singleton Pattern

Implemented in **ProgressLogger** and **ConfigurationHandler** classes to ensure single instance throughout the application, ensuring consistency and reliability.

Observer Pattern

Utilized in ProgressManager and IProgressListener implementations for decoupled progress reporting and modular reaction to progress updates.

Factory Method Pattern

Applied through ICompressionFormat interface for seamless integration of various compression formats into the system.

Abstract Factory Pattern

Utilized in **Compression** abstract class to encapsulate common functionalities of different compression formats, promoting code reusability and consistency.

Application of design patterns

Author(s): Jaïr Telting

	DP1
Design pattern	Singleton (Configuration Handler)
Problem	The problem that would arise if the **ConfigurationHandler** class was not a singleton is that multiple instances of the handler could be created throughout the application. This could lead to inconsistency in managing configuration settings. For instance, different parts of the application might access or modify the configurations independently, resulting in conflicting or outdated values being used.
Solution	With the singleton pattern, only one instance of the **ConfigurationHandler** exists throughout the application's lifecycle. This ensures that all parts of the application access the same instance when retrieving, modifying, or saving configuration settings.
Intended use	At runtime, the singleton instance of the **ConfigurationHandler** class is accessed to manage configuration settings consistently. When users need to retrieve or modify configuration values, it calls the *getInstance()* method of the **ConfigurationHandler**. Furthermore, when retrieving a configuration value, *getProperty()* method is called, which gets the corresponding value from the properties stored internally. Similarly, when updating or adding a new configuration property, *setProperty()* method is utilized, which modifies the properties within the singleton instance. Subsequently, to persist any changes made to the configuration, the application invokes the *saveProperties()* method, which writes the updated properties back to the configuration file.

	DP2
Design pattern	Singleton (Progress Logger)
Problem	If the **ProgressLogger** class were not implemented as a singleton, multiple instances of the logger could be created throughout the application's execution. Consequently, each instance might attempt to access and write to the same log file independently. This could result into inefficient resource utilization.
Solution	By implementing the **ProgressLogger** a singletonclass, ensures the singular existence of the logger throughout its runtime. In designs where multiple instances of the logger could potentially write to the same log file

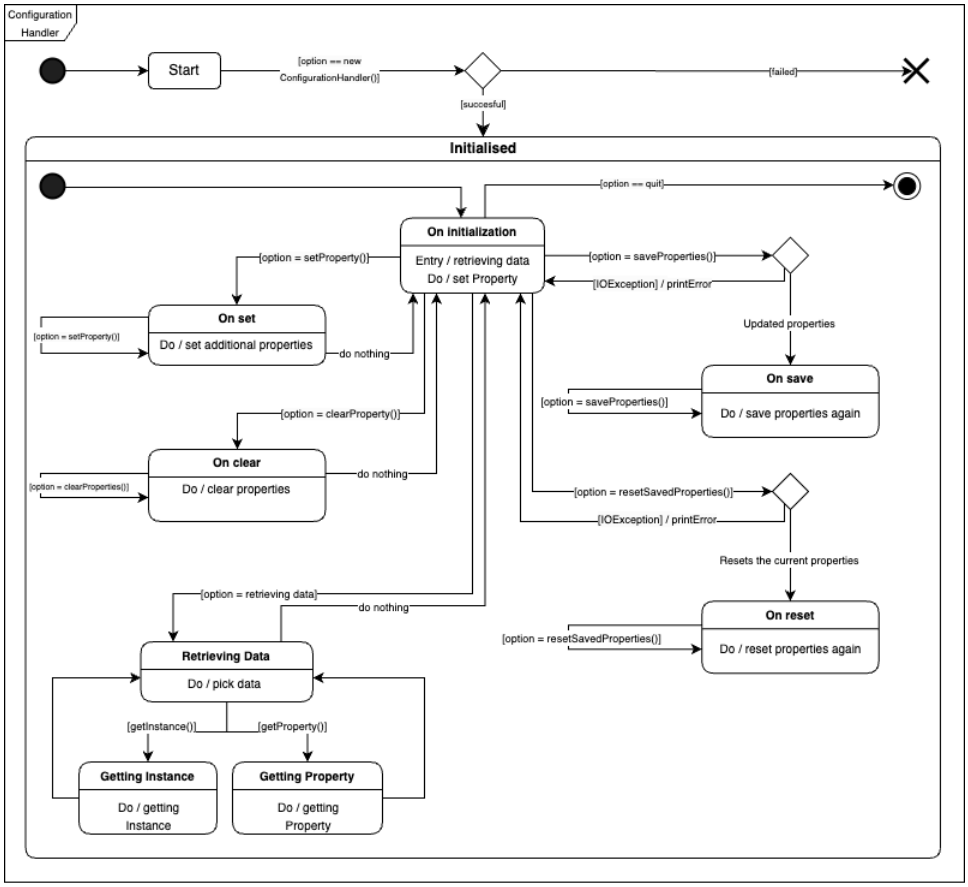
	simultaneously, the risk of data corruption issues is evident. So, by enforcing the presence of only one logger instance, the application prevents this.
Intended use	When a user wants to report progress, the *getInstance()* method of the ProgressLogger class gets invoked to obtain the singleton instance. Once obtained, it calls the *update (Status status, File file)* method on the singleton instance, providing the current status and relevant file information. The **ProgressLogger** then timestamps the progress update and writes it to the log file.

	DP3
Design pattern	Observer (Progress Manager)
Problem	The main problem that arises without a centralized **ProgressManager** is the lack of a standardized way to manage progress updates and notify observers across the application. If each component handles progress updates independently, it can lead to inconsistencies in tracking progress.
Solution	With the Observer pattern, the **ProgressManager** acts as the subject, while the **ProgressLogger** serves as an observer. This design ensures that the **ProgressLogger** receives notifications about progress updates consistently.
Intended use	At runtime, the Observer design pattern is utilized to enable communication between **ProgressManager** as the subject and the **ProgressLogger** as the observer. Initially, the **ProgressLogger** registers itself as an observer with the **ProgressManager** using the *addListener()* method. Subsequently, when progress occurs, i.e. archiving or de-archiving, the **ProgressManager** triggers the *notifyListeners()* method. Upon receiving the notification, the **ProgressLogger** reacts accordingly.

Revised state machine diagrams

Author(s): Simon Vriesema

Configuration Handler



Upon successful instantiation of a **ConfigurationHandler** object, it enters the active/initialized state, enabling users to engage in various operations. Within this state, users can retrieve data using the *getInstance()* and *getProperty()* methods, as well as set properties using *setProperty()*. This flexibility allows users to seamlessly manage configurations, either by fetching existing properties or defining new ones at any point during the active session.

When users request data retrieval, whether for an instance or a specific property, they can expect to obtain the desired information. Even if the instance hasn't been initialized yet, invoking the static methods will gracefully handle the situation, ensuring that the program remains stable. If the instance is null due to not being created yet, an error message is printed, preventing the program from crashing.

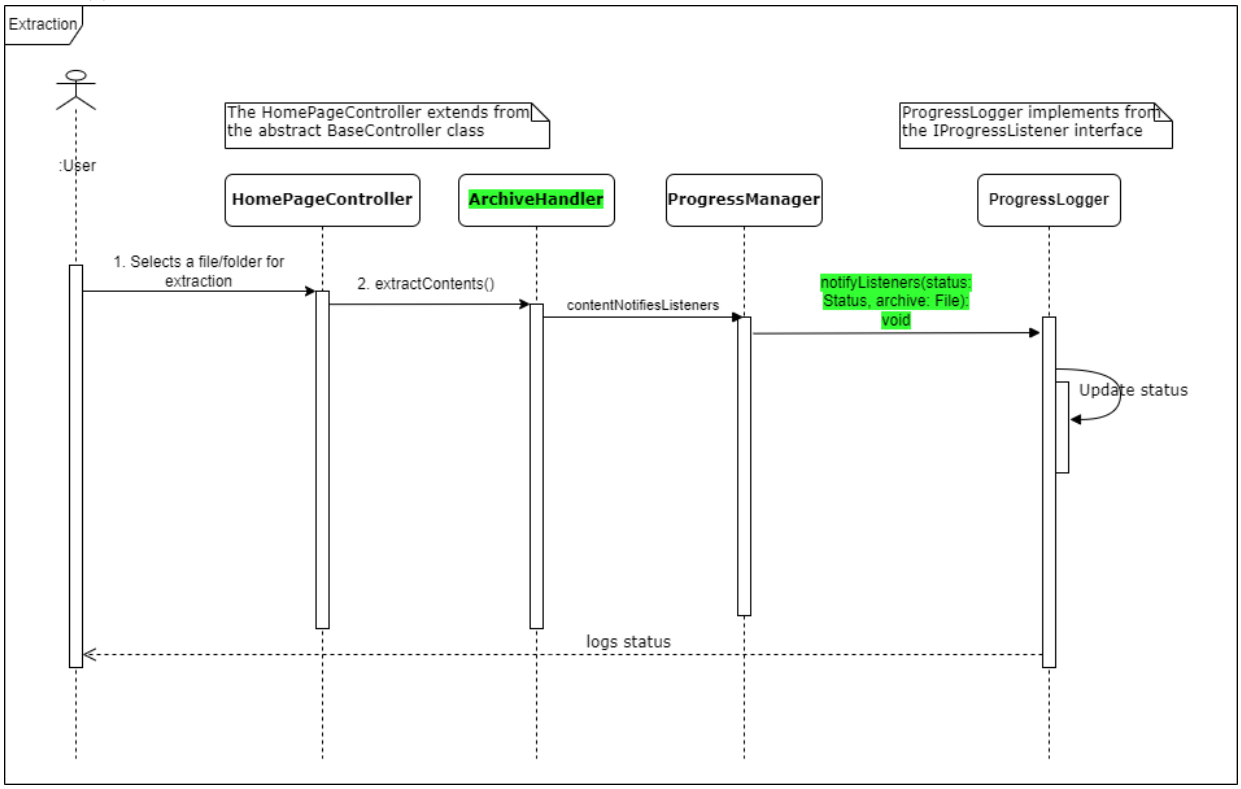
Upon completing their modifications, users can invoke the *saveProperties()* function to persist the changes made during the active session. This action commits all alterations to the configuration, ensuring their preservation for future use. Subsequently, users may opt to conclude this session, transitioning out of the active state.

In addition to the *saveProperties ()*, the **ConfigurationHandler** features the *resetSavedProperties()* function. This function empowers users to revert to the previously saved state, effectively undoing any modifications made during the active session. Like the *saveProperties()*, the *resetSavedProperties()* function may encounter exceptions during execution. However, these exceptions are adeptly managed to prevent any disruptions to the program's operation.

Furthermore, users can clear properties using the *clearProperty()* function, allowing them to remove specific configurations as needed. This additional functionality enhances the versatility and usability of the **ConfigurationHandler**, providing users with comprehensive tools for managing configurations effectively.

Revised sequence diagrams

Author(s): Joli-Coeur Weibolt



The current sequence diagram works the same as the previous one. Based on the changes that we made we changed the name from **contentExtractor** to **ArchiveHandler**. The **ArchiveHandler** is a class that has an *insertContents()* and *extractContents()* functions. The call from the **ProgressManager** was also a little bit different than previous. To clearly see what we have adjusted, the adjustments are highlighted in green in the picture above.

Implementation

Author(s): Jaïr Telting, Dennis Moes

Strategy: from UML to Code

The first step we took to migrate our implementation from its UML representation to actual code was designing and implementing the front end. As shown in our class diagrams, we used JavaFX to implement this. Furthermore, we translated most classes in the diagram to a single file in our project's src directory. We then added each method in the class diagram into its respective classes (no implementation yet). Additionally, relationships present in our class diagram are included in our implementation, i.e. the **HomePageController**, **SettingController**, and **ReportController** extending the **BaseController** class. Note that the class diagram of our system underwent several changes throughout our design process (as seen in the revised class diagram section).

After implementing the front end, we decided to evaluate how different components of our system work with each other. In doing so, we ensured that testing specific functionalities verified the continued successful interaction of linked components within the system. The sequence diagrams implemented in Assignment 2 proved useful in this step.

Key Solutions

Compressing and Decompressing

One key solution we found for implementing compression and decompression in our JavaFX project was to utilize external libraries. While finding a library for ZIP compression (like Zip4j) was relatively straightforward, locating one for GZIP compression proved more challenging. After extensive searching on platforms like Stack Overflow, we stumbled upon a recommendation to use jarchivelib. Despite a broken link, we found the user's GitHub profile and examined their project focused on compression formats in Java.

Studying their project provided valuable insights, and we were able to implement their library effectively into our own system. This solution helped us implement the compression and decompression functionalities into our application.

Encryption

The other key solution of our project revolves around the encryption archives. Beginning with the compression of the selected folder, we ensure optimal storage and transmission efficiency. Thereafter, employing AES encryption, we secure the compressed archive. Importantly, to minimize decryption overhead for accessing metadata, we delay the addition of metadata until after encryption. This approach allows for quick access to essential information without necessitating decryption of the entire archive. By implementing these steps, we achieve enhanced security, optimized performance, and efficient metadata access, facilitated by our AES handler.

Demonstration of the system

A demonstration of our system is shown: here. The flow of execution is as follows:

1. Start of Application:
    a. When *starting the Application (*starts by running *main)*, the user starts at the Home Page. From here, the user has several choices to continue.
        i. The user can either choose to *add a folder:*
            1. Should the user choose to add a folder, the user can *add metadata*, *encryption*, or *archive* the Folder. Additionally, the user can generate a report stating when the archiving is (compiling or) finished.
        ii. The user can either choose to *add an archive:*
            2. Should the user choose to add an archive, the user can de-archive the archive. It should be noted that the compression types the archiver supports are GZIP and ZIP. Additionally, Similar to archiving a folder, the user can generate a report stating when de-archiving is (compiling or) finished.
        iii. The user can open the *settings* menu:
            3. Should the user view the settings menu, the user can adjust the compression format. Furthermore, the user can change the location of compressed and decompressed data (output). Lastly, the user can reset the encryption key.
    b. Additionally, the user can choose to clear the files selected.

Notable locations

Component	Location
Main Java class	....\Software-Design-Archiver\src\main\java\nl\vu\cs\softwaredesign
JAR File	....\build/libs/software-design-vu-2024-1.0-SNAPSHOT.jar

Time logs

Team number	57		
Member	**Activity**	**Week number**	**Hours**
Dennis Moes	Implementing feature 1, 6 and the JavaFX controller class	6, 7 & 8	12
Dennis Moes	Implemented encryption support	6, 7 & 8	8
Dennis Moes	Implemented reflection for compressionhandler	6, 7 & 8	4
Dennis Moes	Implemented default path support	6, 7 & 8	6
Joli-Coeur Weibolt	Implementing feature 4, 5, Revise extraction sequence diagram, working on multiple parts of the final document, creating the readme	6, 7 & 8	30
Jaïr Telting	ProgressLogger implementation	7 & 8	5
Jaïr Telting	Implementation section of report	8	6
Jaïr Telting	Revise Package diagram and add description	8	4
Jaïr Telting	Application of design patterns section	8	2
Simon Vriesema	Explained the class diagram	8	6
Simon Vriesema	Created and explained the State Machine diagram	8	4
Simon Vriesema	Worked on the logging functionality	8	6
Simon Vriesema	Implementing feature 2, 3 and the state machine diagram for ConfigurationHandler		
		TOTAL	93