System Design Document for

MailMe

Authors:

Martin Fredin

Elin Hagman

David Zamanian

Alexey Ryabov

Hampus Jernkrook

date 2021-10-24

version 2.0

1 Introduction

MailMe is a simple email desktop application in which the user can login to already existing accounts from gmail, outlook and hotmail. The main functions of the application are reading, writing and sending emails. There are some additional features which are commonly found in other similar email clients, such as filtering, searching for and adding attachments to emails. However, the purpose of this project is not to compete with already existing email clients, but to construct an application which uses good object-oriented design, is easily extendable and is loosely coupled to the services it uses. This document describes the different levels of the design.

1.1 Stakeholders

The stakeholder of the project is our teaching assistant who has been supervising the project. We want to provide her a new and user-friendly email client that is constructed by using object oriented design.

1.2 Definitions, acronyms, and abbreviations

- **ESP (Email service provider)**: The email server of the email domain provider.
- **POP3**: Short for Post Office Protocol version 3. This is the third version of the Post Office Protocol, which is an application-layer internet standard protocol used by email clients to retrieve email from an email server. (Post Office Protocol, 2021)
- **IMAP (Internet Message Access Protocol:** Is an internet standard protocol for email that allows your email client to get access to email accounts on an email server. (Internet Message Access Protocol, 2021)
- **SMTP**: Short for Simple Mail Transfer Protocol. This is an internet standard communication protocol for sending and receiving emails. (Simple Mail Transfer Protocol, 2021)
- **Port**: A communication endpoint of the computer's operating system. The port number depends on the IP address of the host and the transport protocol used for the communication. (Port, 2021)
- **Host**: A device connected to a computer network and assigned at least one internet address. Hosts may work as servers offering services to users or other hosts on the computer network. (Host, 2021)
- **Attachment**: File sent bundled within an email.
- **Account**: An account existing on a supported ESP.
- **GUI**: graphical user interface.
- **TCP/IP(Transmission control protocol):** Is the set of communications protocols currently used on the internet. (Transmission control protocol, 2021)
- **SSL (Secure sockets layer):** Is a cryptographic protocol designed to provide communications security over a computer network. (Secure Socket Layer, 2021)
- **TLS (Transport layer security):** Is the successor to SSL. It's an improved version of SSL. (Transport Layer Security, 2021)
- **Serialize:** An object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data

stored in the object. (Serialization, 2021)

- **Deserialize:** Back to object from serialized form.
- **MVC (Model View Controller):** A design pattern where the model, view and controller are split up. (Model View Controller, 2021)

2 System architecture

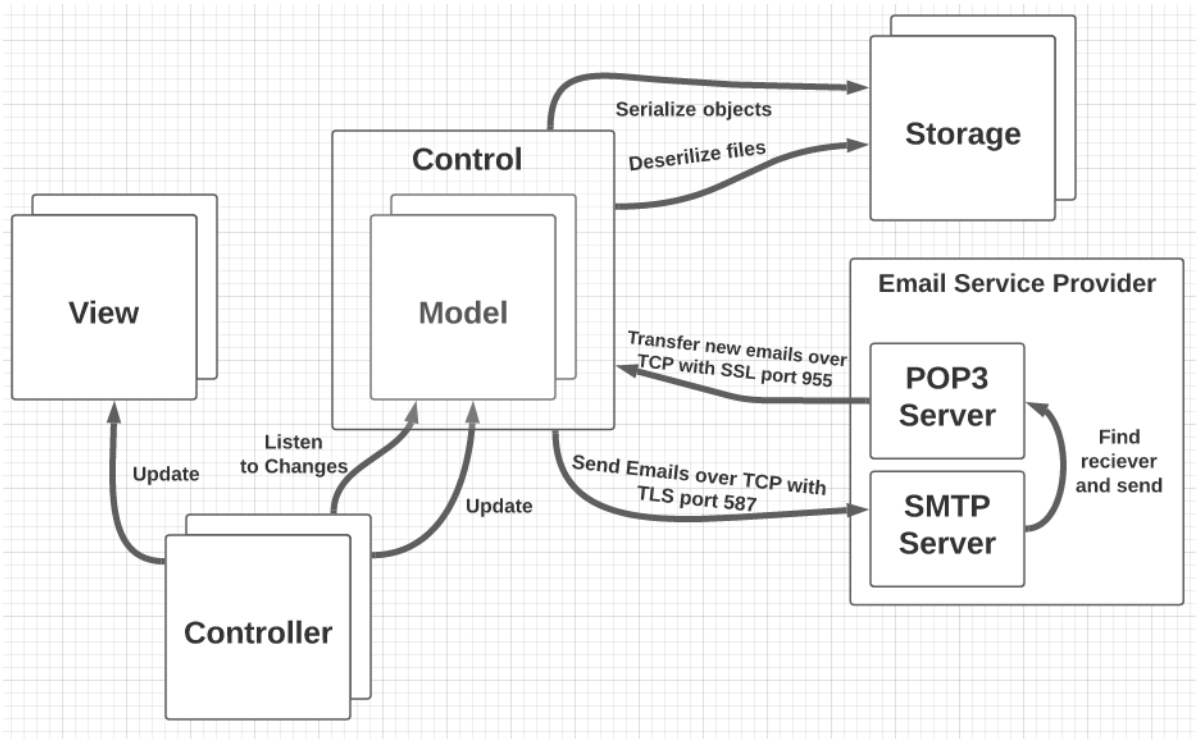

Figure 1. High level design model

The application uses the MVC variant passive view (Fowler, 2006), because it is the most appropriate when working with JavaFx's fxml-files and SceneBuilder. The main difference between other variants of MVC is that the view contains essentially no logic. These responsibilities are handed off to the controller, thereby eliminating the dependency between View and Model.

Following are the responsibilities of the MVC components. The model's responsibilities is to keep the data and state of the application at runtime, and to make changes to data which the controllers later can react to. The view is passive and thereby its sole responsibility is to serve the content which is determined statically and dynamically – by the controller. The controller needs to act as a mediator between Model and View. It needs to handle user input from the view and then update the model's state accordingly. Furthermore, it needs to listen to state changes in the model's state and update the view to reflect the changes. The application extends the passive view pattern with a control layer which is placed in front of the Model. The first responsibility of Control is to decouple Model and Services. The second responsibility is to manage exceptions from the Model, e.g. AccountAlreadyExistsException and EmailDomainNotSupportedException, and to manage exceptions from the services, e.g. OSNotFoundException and ServerException.

The application has two services. The first one is the storage service. It is responsible for how data is stored, and the storing process. The Model is very unopinionated in how the

data is stored, therefore it is very easy to replace the concrete storage solution. The current implementation uses a local storage solution, which saves java objects on the computer via serialization and retrieves the objects using deserialization.

The second service is the email service provider (ESP) service. It is responsible for all communication with the email servers, and to parse the response. To send emails the ESP service connects and sends the emails to an SMTP server. The connection is over TCP/IP and uses the SMTP protocol with TLS encryption (port 587) for the data transfer. After the email is sent to the SMTP server, the email is forwarded to the correct recipient's email provider. To receive emails the ESP-service downloads the emails from the POP3 server, which is connected through TCP/IP using the POP3 protocol with SSL encryption on port 995.

The application uses the POP3 protocol over its successor IMAP. The reason for this is because the application aims to provide an unifying experience irrelevant if the email provider is for example Gmail or Outlook. By using POP3 all the connected accounts will have the same set of folders and features. Another advantage is that only new Emails are downloaded from the server, which favors users with slow internet connections. Lastly, it appeals to users who don't want their emails stored on Google or Microsoft's servers, and would rather have them locally for privacy reasons.

3 System design

This section provides a closer look at the individual packages. Every package is explained briefly, and design principles and patterns are presented and motivated.

3.1 View

The resource package includes the view package together with a package for images and icons.
The view package consists of all of the individual fxml views with the Master as the “main view”. Each view has their own css file for styling and there is also a DefaultStyle file used by almost all of the views that includes standard colors, styling for buttons, text fields, text areas etc to avoid repeating code.
Each view is represented by a controller that loads the view either in an already existing pane in the masterView or by creating a new stage in the windowOpener and loading the view onto a new window.

3.2 Controller

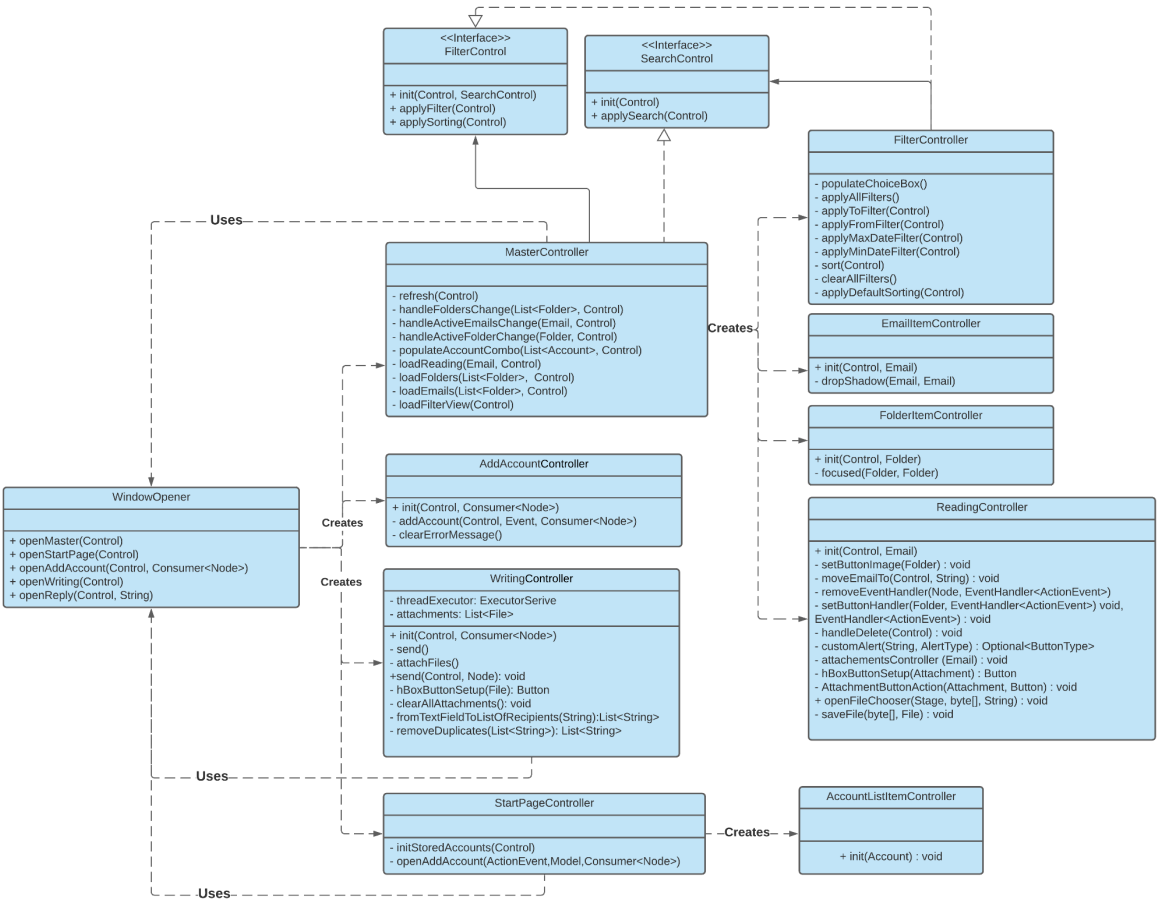

Figure 2. UML diagram of the controller package

The Controller package consists of a static class named WindowOpener, two interfaces and the controllers. The WindowOpener class contains static methods that can be called to initiate a Master-, AddAccount-, Writing- or StartPageController and open their corresponding view in a separate window. In this way these controllers can be decoupled from each other.

The controllers get access to the logic and state of the model through dependency injection. Each controller that uses the model's logic takes an instance of a Control as an argument in its init method. Dependency injection was used rather than the Singleton pattern to be able to protect the state of the model from unexpected changes. However, since Control is present in almost every controller it could have been argued that the Singleton pattern could have been used as well.

Observer pattern is used to update the controllers of changes in the model. In adherence with favor composition over inheritance, the application uses a compositional observer pattern. Therefore, Model contains several observable attributes, e.g. activeAccount and folders, which are responsible for handling its observers rather than the model class itself. The controllers do not implement an Observer interface, instead, to subscribe to one of the observables the controller gives a lambda expression as argument in the corresponding

addObserver method. If a controller (or something else) updates one of the observable attributes all the observers of this attribute will be notified and handle the change accordingly.

3.3 Model and Control

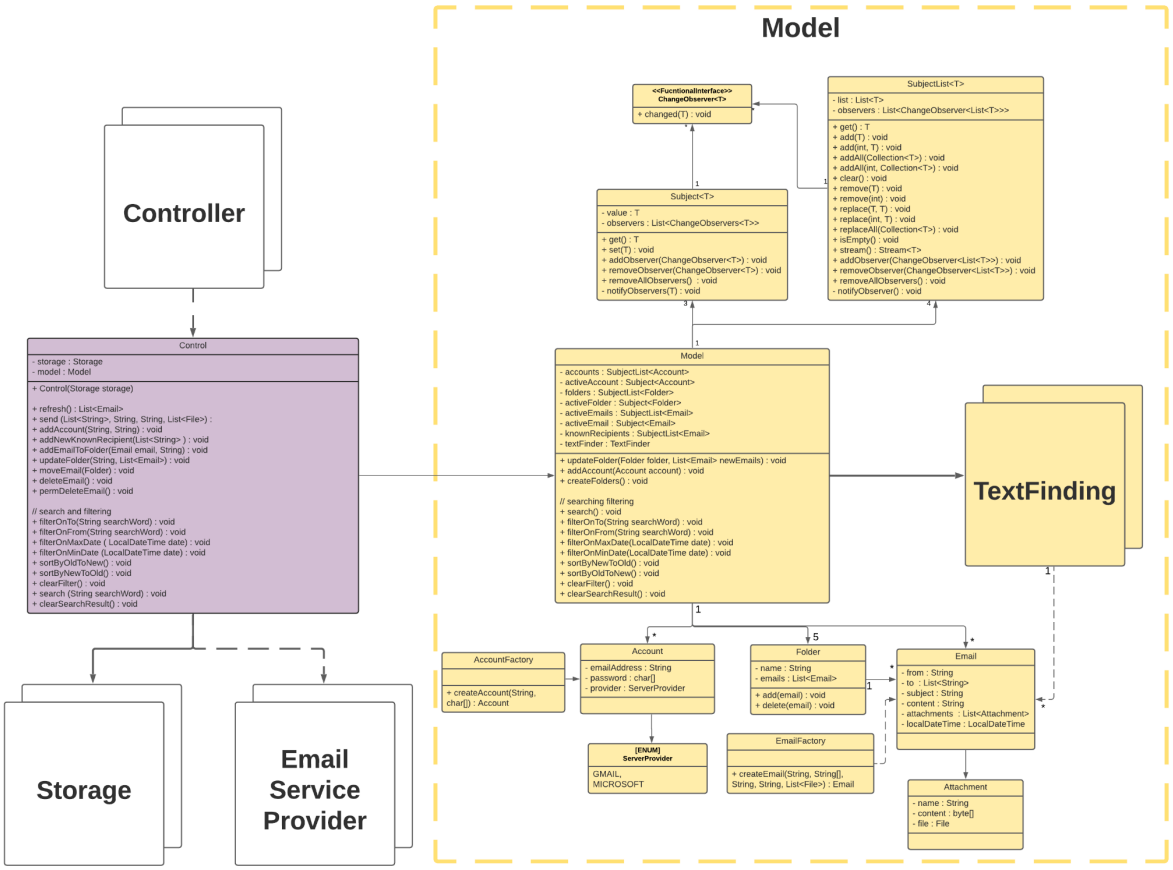

Figure 3. Package Model with relations to other packages.

The Model consists of several domain classes, e.g. Account, Folder, and Email. These classes are used in every layer of the application, and are represented in the domain model (see Figure 7).

AccountFactory and EmailFactory are instances of the factory pattern. The application uses factory pattern to encapsulate the creation of objects, in adherence to single responsibility principle (SRP). AccountFactory is used to hide the specifications of how the ServerProvider field is determined. EmailFactory is used to set the correct date and attachments. The Folder class doesn't require a factory because the default constructor is sufficient.

The Model class holds the state of the application and acts as interface for the Model package, in adherence to the facade pattern and dependency inversion principle (DIP). The application state consists of seven observable fields (see Model class Figure 3). The observables utilise the observer pattern to notify decoupled observers of changes in the state. This enables Model to be decoupled from View and Controller which is highly favorable according to the MVC pattern.

The observables are called Subject and SubjectList in the model, and the observers are called ChangeObservers. In adherence with favor composition over Inheritance, the application uses a compositional observer pattern. It leverages all the pros of a “coupled” observer pattern without the cons (such as poor scalability and hard to maintain) (Wikipedia). This is because the domain classes (Account, Folder, Email) are not aware of the observers, which is accomplished by encapsulating the observable functionality in a separate class called Subject. In doing this, the pattern adheres to separation of concern, because the domain classes don’t concern themselves with adding, removing, and notifying observers. The observers are also implemented using composition. The design of the observer pattern is inspired by the javafx.beans library, specifically SimpleObjectProperty, ObservableList, and ChangeListener.

The Control class is not part of the Model, but because it handles the models relations it is appropriate to include it in this section. Control is an extra layer on top of the model (see Figure 1), and acts as an glue between controllers, model, and services. It is responsible for decoupling model and services and handling exceptions between the different parties. Thus, it adheres to separation of concerns (SoC). It is an instance of the facade pattern, but also acts similarly to the Mediator Pattern (although not implemented as such).

3.3.1 TextFinding

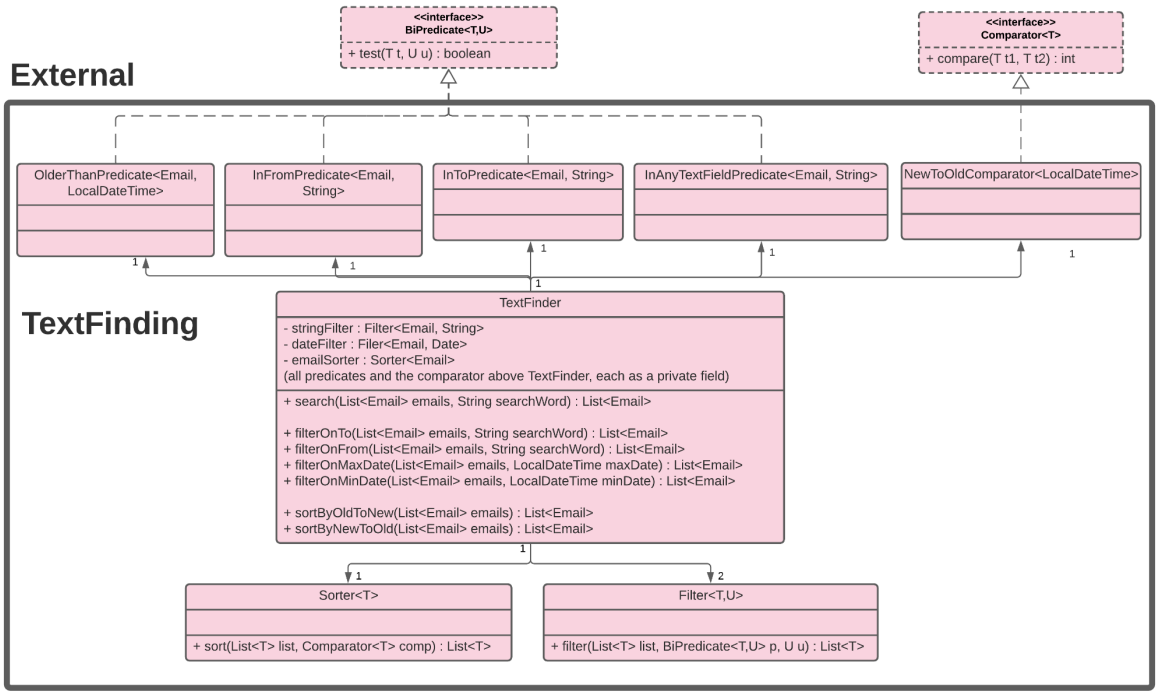

Figure 4.

TextFinding is responsible for the backend part when the user applies a search, filter or sorting, selecting the exact subset of emails within the current folder that fulfil the user's applied conditions.

The TextFinding package is communicated with by external classes (in our case only Model) via a facade pattern. As shown in the above image, TextFinding consists of many small classes, but external classes may only access the public class TextFinder. All other classes are package private. Upon calling methods on a TextFinding instance *i*, *i* delegates to the appropriate classes from the remainder of the package, combining them in methods as appropriate.

The main algorithms of TextFinding – sort(...) and filter(...) – are generically implemented to achieve a higher reusability of these components. They also enable a higher extensibility, by allowing more BiPredicates and Comparators to be defined and used with these algorithms, if wanting to add additional filter and sorting alternatives.

3.3 Service

The application uses two services which are described below.

3.3.1 Email Service Provider

The email service provider uses factory pattern and consists of four classes: EmailServiceProviderFactory, abstract class EmailServiceProvider and GmailProvider, MicrosoftProvider, see Figure 5 below. It is responsible for the connection to the server and sending/parsing an email. In this way a new provider can be easily added into the application and expanded. EmailServiceProviderFactory is an abstract class, its function is to determine which email service provider is going to be used depending on the user's account details. Both GmailProvider and MicrosoftProvider extend EmailServiceProvider class which is an abstract super class to the two. It is responsible for testing/verifying connections, parsing information e.c. GmailProvider and MicrosoftProvider are responsible for esp specific properties and sending an email.

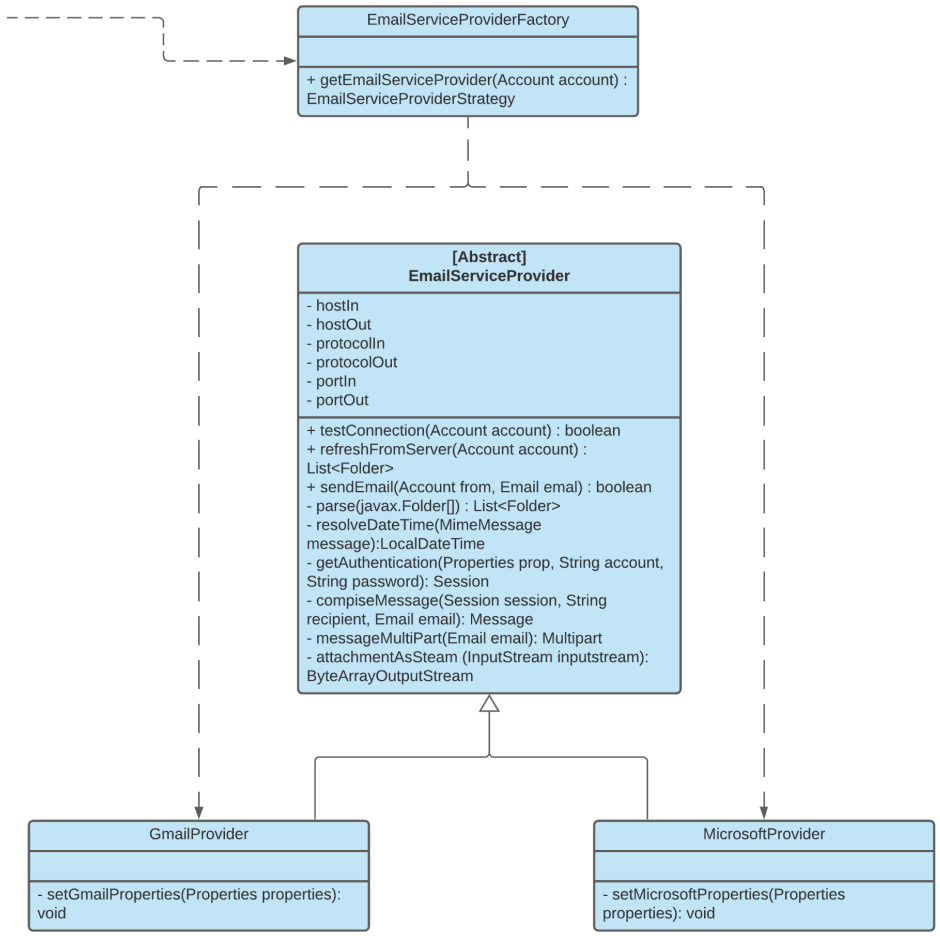

Figure 5.

3.3.2 Storage

Storage is responsible for storing user application files on the machine. It consists of a Storage interface, LocalDiskStorage and OSHandler. There are also three custom exception classes: StorageException, OSNotFoundException, AccountAlreadyStoredException, see Figure 6 below. Those are specific for storage. LocalDiskStorage implements the Storage interface and is responsible for storing account information. This could be changed to another solution in the future. OSHandler sets the application directory depending on the user's operating system. It serves as a helper class.

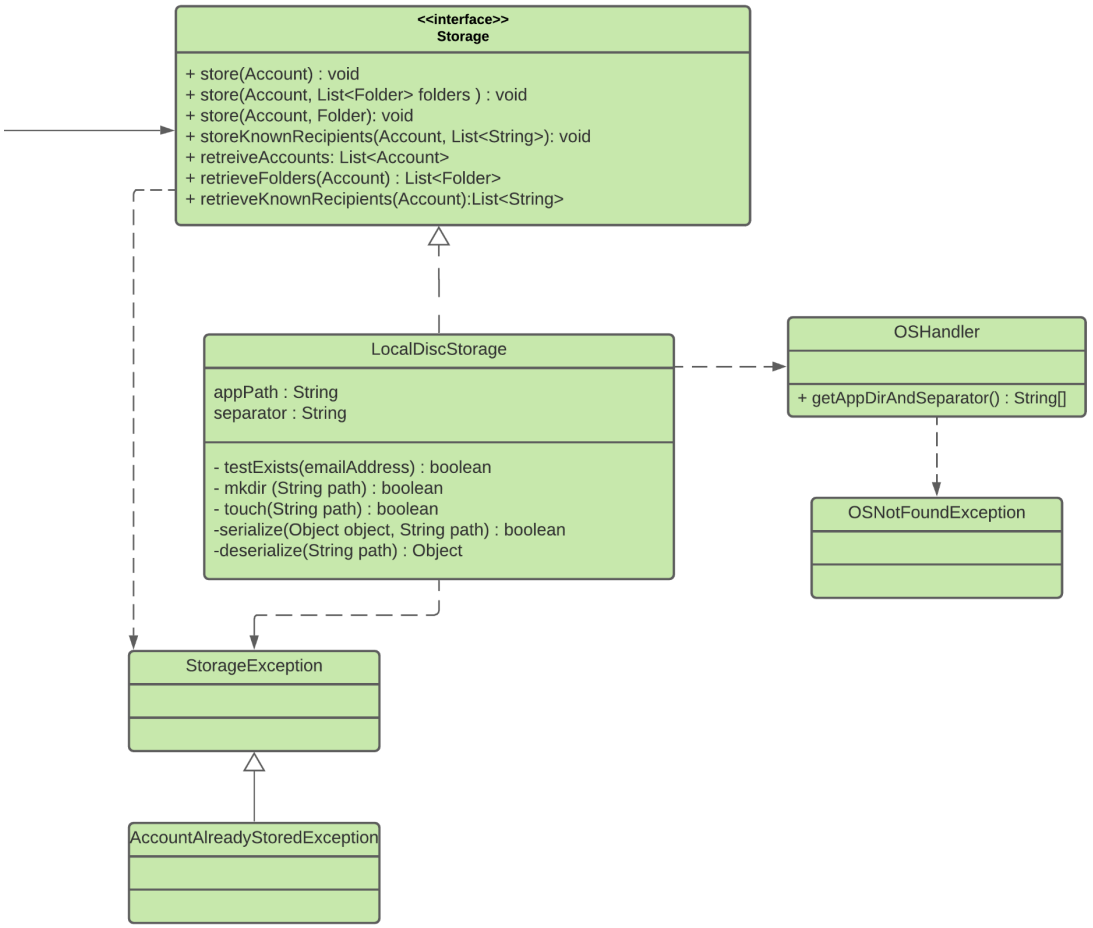

Figure 6.

4 Persistent data management

We decided to use local storage as the location for storing all of our emails, users, images and icons.

Depending on the ESP, we make a request for all the folders and emails from their database and store it locally on the user's computer. We create a local directory with a path depending on the OS (operating system) of the user's computer.

Images and icons are stored in a folder called ImagesAndIcons, also locally.

The reason for storing the folders, emails and users locally is performance related, we don't have to retrieve all the files from the ESP everytime we open the application. Only when we add a new account or refresh for new emails. And we don't have to add a new account or log in everytime we start the application.

One potentially huge disadvantage with storing the users data locally is that it causes a big security risk due to the users email and password being stored in a plain text file on the computer. We considered hashing the password, but this would not work with our application as we need to use the plain text password when connecting to the servers.

The app directory is called *Group77* and within this parent directory the application stores at level:

1) Directories with name equal to the email address of the account it stores information for. This guarantees unique directory names.
2) Folders and recipient-suggestions for the given account represented by the parent directory (at level 1).

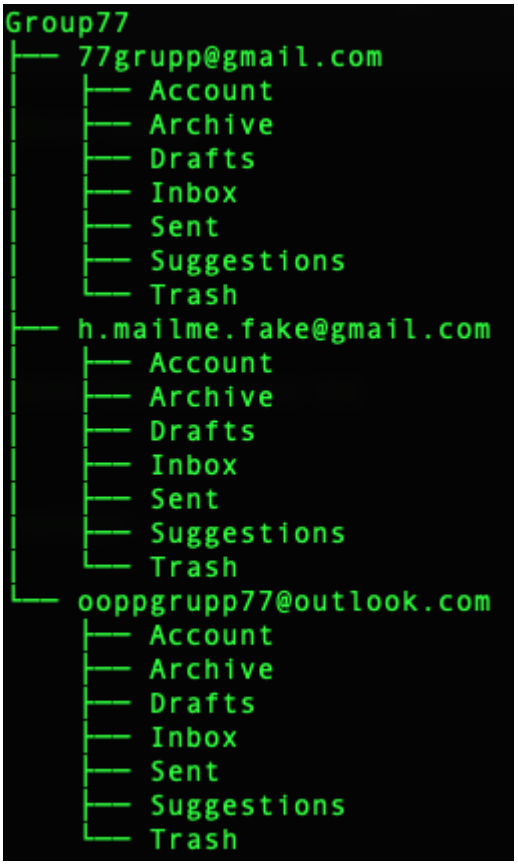

Figure 7. Directory structure of the application directory. Example with three added accounts.

5 Quality

Our application, by design, is divided into several high-level packages/modules: model, services, controller and view (fxml resources). The model package is unit tested. For each class or functionality in this package there exists a test class. The line coverage of the tests of the model package is at 95%.

The stakeholders only require the model to be unit tested because the view is obviously hard to test using unit tests, and the service package contains a lot of input and output (IO) which makes tests unreliable.

Junit tests are located in the folder *test* within the *src* folder under the the main folder of the OOPP-Group77 project:

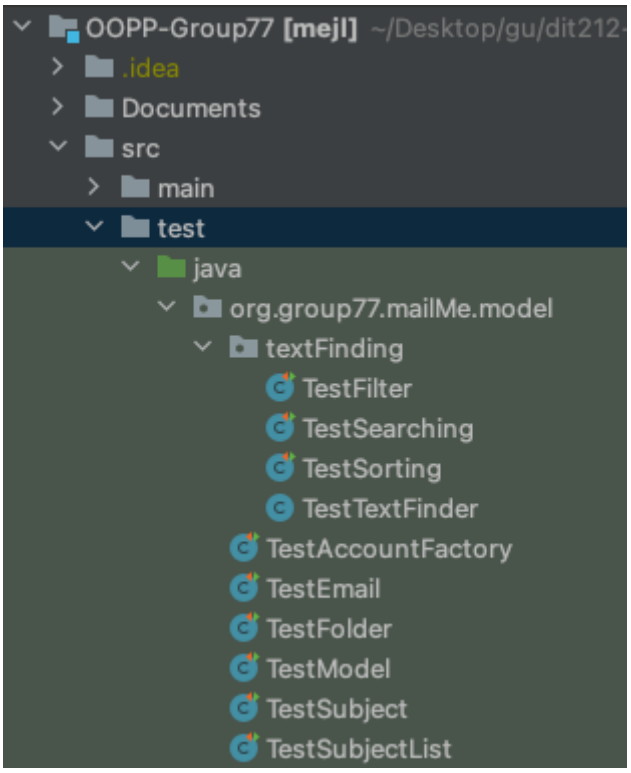

Figure 8. All junit test classes are located within the test folder. Test classes are divided by the package they belong to and contain tests for classes within those packages.

We have during the project tried to use continuous integration via Travis, but there have been many issues with this service. First, it only ever ran one test class and not all within the *test* package. We could not get this to work properly. Then, after a migration from one branch to another due to a design change, and a change of Java version, Travis continuously gave build errors for each push to GitHub. This too remains an unsolved issue. Travis CI link: https://app.travis-ci.com/github/hjernkrook/OOPP-Group77

We have also tried to use the analytical tool PMD to work and use this to clean up our code but PMD also gives us errors. The currently blank report in pmd.html that gets generated by *mvn site* currently gives for each file *f*:

```
PMDException: Error while processing f
net.sourceforge.pmd.PMDException: Error while processing f ...
Caused by: java.lang.IllegalArgumentException: Unsupported class file major version
60 ...
```

JDepend, on the other hand, works. Among other things, it reports that we have no cyclic dependencies and gives a rather high Instability-value for most packages, which is concerning of course.

List of all known issues:

1) Sending an email from user A and changing account to user B before the email is fully sent leads to the email being stored under B's 'Sent', not A's as is intended.
2) Archiving draft d and then pressing the "restore"-button on d in 'Archive' leads to d being stored in 'Inbox' instead of being moved back to 'Drafts'. This also holds for sent emails, moved to 'Archive' from 'Sent'.
3) Most of the time, the application does not terminate correctly upon closing the window. The process needs to be aborted using ctrl+C from the command line. We believe this is due to some thread continuing to run in the background.
4) The HTML part of HTML-emails with attachments do not show properly. We only get an object string representation of the HTML content. The attachments are there though. This means that not even emails with only a simple plain text content and attachments, sent from the ordinary gmail web client, show the content properly.

5.1 Access control and security

The access control of our application consists of login information. To be able to use the MailMe application, the user has to enter/login with its already existing email address and password. The login credentials are checked against the appropriate server to determine whether the account actually exists and should be added to our client.

The user only has to login once, the application will then either
1) select the only submitted account as the active one automatically, or
2) prompt the user to select one of several submitted accounts upon launching the application.

5.2 Potential improvements and reflections

During the project, we have gained both deeper knowledge in object oriented programming, as well as of email clients and their design. Due to lack of knowledge and a clear enough picture of what the end result should exactly be, some pitfalls were fallen into. The most evident concerns password security. Currently, Account instances are exposed towards the frontend, allowing the frontend free access to account passwords. Had we had more time to address this issue we would have implemented an AccountWrapper class, which wraps an Account by only exposing the email address to external classes. We would then have used this as the publicly exposed part of the Model and only used the corresponding Account instances in the backend when needed.

Some other potential improvements and reflections concern:

- Exceptions handling
    - Control should handle more exceptions and let less exceptions through to the controllers. Control does handle a reasonable amount of exceptions but it should probably be more.
    - The application should always use the exception as an argument when rethrowing an exception of a different type.
    - Use different catch blocks when dealing with multiple exceptions.
    - We use exceptions instead of conditionals as goto statements.
- Security
    - Adding to the password security, it would have been good to be able to encrypt Account instances' password before storing them and decrypt only when they needed to be sent to the server. Since this form of security was not a key focus of this project, this was deprioritized.
- MVC
    - The Model class contains some fields which may be seen as part of the View's state and not so much the model. This include the fields
        1) activeFolder: this is the currently selected folder,

2) activeEmail: this is the currently selected email,
3) activeEmails: this is some subset of the emails in activeFolder.

This makes the model inflexible because it assumes that only one folder and email is opened at a time. Consequently, this can limit the ways the GUIs connected to the model can look like.

6 References

Maven. Available at: https://maven.apache.org/index.html (Accessed. 07 10 2021)

JavaFX. Available at: https://openjfx.io/ (Accessed: 07 10 2021)

JavaMail API https://javaee.github.io/javamail/docs/api/ (Accessed: during september, October 2021)

Apache Commons. Available at: http://commons.apache.org/ (Accessed: during september, october 2021)

Fowler, M. (2006) Passive View. Available at: https://martinfowler.com/eaaDev/PassiveScreen.html (Accessed: 06 10 2021)

Post Office Protocol. Available at: https://en.wikipedia.org/wiki/Post_Office_Protocol (Accessed: 07 10 2021)

Internet Message Access Protocol. Available at: https://sv.wikipedia.org/wiki/Internet_Message_Access_Protocol (Accessed: 07 10 2021)

Simple Mail Transfer Protocol. Available at: https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol (Accessed: 07 10 2021)

Simple Mail Transfer Protocol. Available at: https://en.wikipedia.org/wiki/Port_(computer_networking) (Accessed: 07 10 2021)

Host. Available at: https://en.wikipedia.org/wiki/Host_(network) (Accessed: 07 10 2021)

Port. Available at: https://en.wikipedia.org/wiki/Port_(computer_networking) (Accessed: 07 10 2021)

Internet Protocol Suite. Available at: https://en.wikipedia.org/wiki/Internet_protocol_suite (Accessed: 07 10 2021)

Transport Layer Security: Available at: https://en.wikipedia.org/wiki/Transport_Layer_Security (Accessed: 07 10 2021)

Secure Socket Layer. Available at. https://sv.wikipedia.org/wiki/Secure_Sockets_Layer (Accessed: 07 10 2021)

Model View Controller. Available at: https://sv.wikipedia.org/wiki/Model-View-Controller (Accessed: 07 10 2021)

Serialization. Available at: https://www.tutorialspoint.com/java/java_serialization.htm (Accessed: 07 10 2021)

ControlsFX. Available at: https://github.com/controlsfx/controlsfx (Accessed: 15 10 2021)

7 Appendix

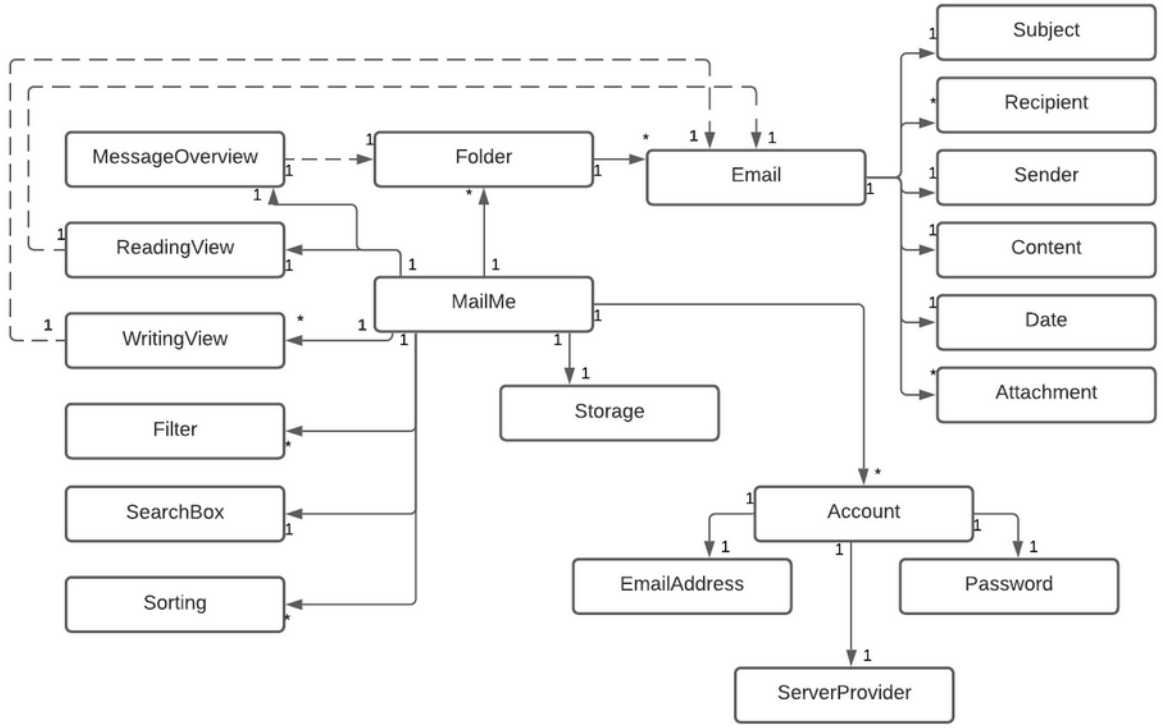

Figure 9. Domain Model