Requirements and Analysis Document for MailMe

Authors:

Martin Fredin

Elin Hagman

David Zamanian

Alexey Ryabov

Hampus Jernkrook

date 2021-10-24
version 2.0

1 Introduction

1.1 Purpose of application

The purpose of this project is to create a simple email client. The application will support basic features that can be found in similar email clients.

1.2 General functionality of application

The application will be a standalone cross-platform desktop application with a graphical user interface (GUI) and will be able to run on Linux, Windows and MacOs.

With our application, users will be able to connect with an already existing email account to an email service provider to receive, read and send emails. There will be additional options to reply to and forward received emails, as well as attach files to emails.

1.3 Scope of application

The application intent is not to be able to compete with already existing email clients similar to ours such as Thunderbird or the one offered by Apple. The application should be able to support simple email client features. The key focus will be on constructing a good object-oriented design for the application.

1.4 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.

1.5 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)
- **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.

2 Requirements

2.1 User Stories

Green annotation means that a user story is fully implemented. Yellow means it is partially implemented (acceptance criteria may have been added as we discovered certain aspects of the application). Red means that the user story or acceptance criteria is not implemented at all (at least not so that it can be used in the application by the user). Orange means that an acceptance criteria is almost met, but may fail occasionally due to bugs.

Green - Implemented.

Yellow - Partially implemented.

Red - Not implemented.

Orange - Implemented but bugs hinder usability.

#User story
Story Identifier: add gmail account
As a: user
I want to: be able to login to my existing gmail account.
So that: I can use my existing account.

Acceptance:

- User is able to submit an account with account details (email and password) upon launching the application.
- User is able to enter and submit a new email address and password for an existing account in a separate addAccountView.
- User will get feedback if the email was successfully connected.
- User will get feedback if the email could not be added due to rejection from the server.
- User will get feedback if the email provided is not supported by the application.

#User story
Story Identifier: login only once
As a: user
I want to: only have to submit my account details once
So that: I can use my submitted account without having to login to it every time I launch the application.

Acceptance:

- The user's account details are stored locally on the user's computer and read by the application when necessary.
- If the account has already been added and the user tries to add it again, the application's stored info connected to the submitted account should not be

overridden, in order to keep other data already saved and linked to the account.

#User story
Story Identifier: choose active account
As a: user
I want to: have a list of my connected accounts.
So that: I can select one as active.

Acceptance:
- The user gets to choose from a list of added accounts when launching the application.
- User is able to get a drop-down of added accounts.
- User is able to select an account in the drop-down.
- The active account state is updated accordingly.
- User receives visual feedback of which account is active

#User story
Story Identifier: email overview
As a: user
I want to: have a list of previews of all emails in the currently selected folder.
So that: I can get an overview of my email feed.

Acceptance:
- User should be able to see a list of emails displaying the sender, subject and date.
- The list should contain all emails within the currently selected folder.
- The list should be retrieved from the collection of emails stored in the application's storage.

#User story
Story Identifier: writing emails
As a: user
I want to: be able to write a new email to any other email user with any other email address.
So that: I can communicate with other email users.

Acceptance:
- User should get a new window to write the new email in.
- User will be able to input the recipient of the new email.
- User will be able to input the subject of the new email.
- User will be able to input the content of the new email.
- User will be able to send that particular email.
- The 'from' field of the written email is automatically set to the user's active account's email address.
- User should get a notification if the email was successfully sent.
- User should get a notification if the email could not be sent.
- The sent email should be stored in the application's storage under the 'Sent'-folder.

#User story
Story Identifier: email folders
As a: user
I want to: see a list of my email folders.
So that: I get an overview and can navigate between them.

Acceptance:
- User sees a panel with the different email folders.
- User is able to navigate between the folders by clicking on them.
- Clicking on a folder should update the email overview accordingly, with the emails contained in that folder.
- The folder button should be linked to a list of emails, or else retrieve the emails from storage.

#User story
Story Identifier: read emails
As a: user
I want to: be able to read my emails.
So that: I can be up-to-date with my emails.

Acceptance
- User will be able to select a specific email from many emails within an email overview.
- User will be able to open specific emails and read their content with a new reading view.
- The single email button should either be linked to a specific Email object or else retrieve the corresponding email from storage.
- The application is able to show HTML content
- The application is able to read and receive attachments

#User story
Story Identifier: refresh from server
As a: user
I want to: be able to refresh my inbox.
So that: I can see new ones that I have received.

Acceptance
- User will be able to request a refresh from the server by pressing a button.
- The overview with the emails in the inbox should be updated with the emails received from the server, in addition to any emails that were previously in the folder.
- The storage is updated with the newly fetched emails, in addition to any previously stored ones.

#User story
Story Identifier: attach files
As a: user
I want to: be able to attach files to my emails that I am writing
So that: I can send files to other email users.

Acceptance
- User can press an ‘Attach’-button in the writing view to initialize the attach-process.
- Upon pressing the ‘Attach’-button, the user can choose files to attach to the email from their file explorer.
- Emails with attachments can be sent via the server.
- In the ‘Sent’-folder, the user can see what files were attached to a given email when reading it.
- Emails can be stored with information about potential attached files.
- User receives visual feedback that the file is attached

#User story
Story Identifier: save drafts
As a: user
I want to: be able to save email drafts
So that: I can start composing an email and continue to write it at a later time.

Acceptance
- User can click a button to save an email-in-progress in ‘Drafts’.
- User is notified that the currently written email was saved in ‘Drafts’ when closing down a non-empty email without first sending.
- The email is stored in storage under folder ‘Drafts’.
- The user is able to click an email in the ‘Drafts’ overview to continue writing on that email.

#User story
Story Identifier: filter on to
As a: user
I want to: be able to filter my emails based on a specific to-address
So that: I can quickly find specific emails.

Acceptance
- User can enter a text within a to-textfield for the filter.
- User is notified about the outcome of the filtering process.
- User can see what filters are currently applied.
- User can clear the filter.
- The overview displays exactly those emails within the selected folder that fulfil the active filtering condition (user-entered substring is part of one of the to-addresses).
- The overview displays all emails within the selected folder upon clearing the filter.

#User story
Story Identifier: filter on from
As a: user

I want to: be able to filter my emails based on a specific from-address
So that: I can quickly find specific emails.

Acceptance

- User can enter a text within a from-textfield for the filter.
- User is notified about the outcome of the filtering process.
- User can see what filters are currently applied.
- User can clear the filter.
- The overview displays exactly those emails within the selected folder that fulfil the active filtering condition (user-entered substring is part of the from-address).
- The overview displays all emails within the selected folder upon clearing the filter.

#User story
Story Identifier: filter on date
As a: user
I want to: be able to filter my emails based on a given time interval
So that: I can quickly find specific emails.

Acceptance

- User can click on one of multiple max dates to apply the date filter.
- User is notified about the outcome of the filtering process.
- User can see what filters are currently applied.
- User can clear the filter.
- The overview displays exactly those emails within the selected folder that fulfil the active filtering condition (the email's received-date is less than or equal to the user-entered max date).
- The overview displays all emails within the selected folder upon clearing the filter.

#User story
Story Identifier: sort new to old
As a: user
I want to: be able to sort my emails by newest date to oldest date
So that: I can quickly find specific emails.

Acceptance

- User is able to click a button/toggle to make the emails be sorted.
- User is notified about the outcome of the sorting process.
- User can see what sorting condition is applied currently.
- User can clear the sorting condition.
- The overview displays exactly those emails within the selected folder that fulfil the sorting condition.
- The overview displays all emails within the selected folder, in any arbitrary order, upon clearing the sorting condition.

#User story
Story Identifier: sort old to new
As a: user
I want to: be able to sort my emails by oldest date to newest date
So that: I can quickly find specific emails.

Acceptance

- User is able to click a button/toggle to make the emails be sorted.
- User is notified about the outcome of the sorting process.
- User can see what sorting condition is applied currently.
- User can clear the sorting condition.
- The overview displays exactly those emails within the selected folder that fulfil the sorting condition.
- The overview displays all emails within the selected folder, in any arbitrary order, upon clearing the sorting condition.

#User story
Story Identifier: search
As a: user
I want to: be able to search for a specific sub-word within the emails
So that: I can quickly find specific emails.

Acceptance

- User can enter a search word into a search box.
- User is notified about the outcome of the searching process.
- User can see what search word is applied currently.
- User can clear the search box.
- The overview displays exactly those emails within the selected folder that match against the search word.
- The overview displays all emails within the selected folder upon clearing the search box.

#User story
Story Identifier: add microsoft account
As a: user
I want to: be able to login to my existing microsoft account (hotmail, outlook).
So that: I can use my existing account.

Acceptance

- User is able to submit an account with account details (email and password) upon launching the application.
- User is able to enter and submit a new email address and password for an existing account in a separate addAccountView.
- User will get feedback if the email was successfully connected.
- User will get feedback if the email could not be added due to rejection from the server.
- User will get feedback if the email provided is not supported by the application.

#User story
Story Identifier: delete email
As a: user
I want to: be able to delete existing email.
So that: I can have less emails in my folders.

Acceptance

- User is able to delete existing email from their respective folder.

- User is able to permanently delete email from their Trash-folder.

#User story
Story Identifier: move email
As a: user
I want to: be able to move existing email.
So that: I can have better sorting of my emails.

Acceptance
- User is able to move existing email from their respective folder to the Archive-folder.
- User is able to undo moving emails to ‘Archive’.

#User story
Story Identifier: reply to received
As a: user
I want to: be able to reply to received emails
So that: I can quickly answer other email users that have contacted me.

Acceptance
- User can press a button to reply to the received email.
- A writing view is opened upon pressing the reply button.
- The ‘to’ field of the reply-email is automatically set to the sender of the received email.
- The ‘subject’ field of the reply-email is automatically set to the subject of the received email.
- The content of the received email is concatenated with the content of the user’s reply.
- The functionality in other regards adheres to the same acceptance criteria as writing emails in general.

#User story
Story Identifier: forward received
As a: user
I want to: be able to forward my received emails
So that: I can quickly pass along a message to another email user.

Acceptance
- User can press a button to forward a received email.
- A writing view is opened upon pressing the forward button.
- The ‘subject’ field of the reply-email is automatically set to the subject of the received email.
- The content of the received email is concatenated with the eventual content of the user’s additional message.
- The functionality in other regards adheres to the same acceptance criteria as writing emails in general.

#User story
Story Identifier: delete account
As a: user
I want to: be able to delete an account from the application's storage (not server)
So that: I can clean up among my used accounts.

Acceptance

- User can click a button to delete a given account within the application.
- User is notified of the outcome of the deletion of the account.
- The deleted account is removed from the application's storage, along with all emails and other data connected to that account.
- If the deleted account was the active one, the user is prompted to select another active account or add a new account to use.

2.2 Definition of Done

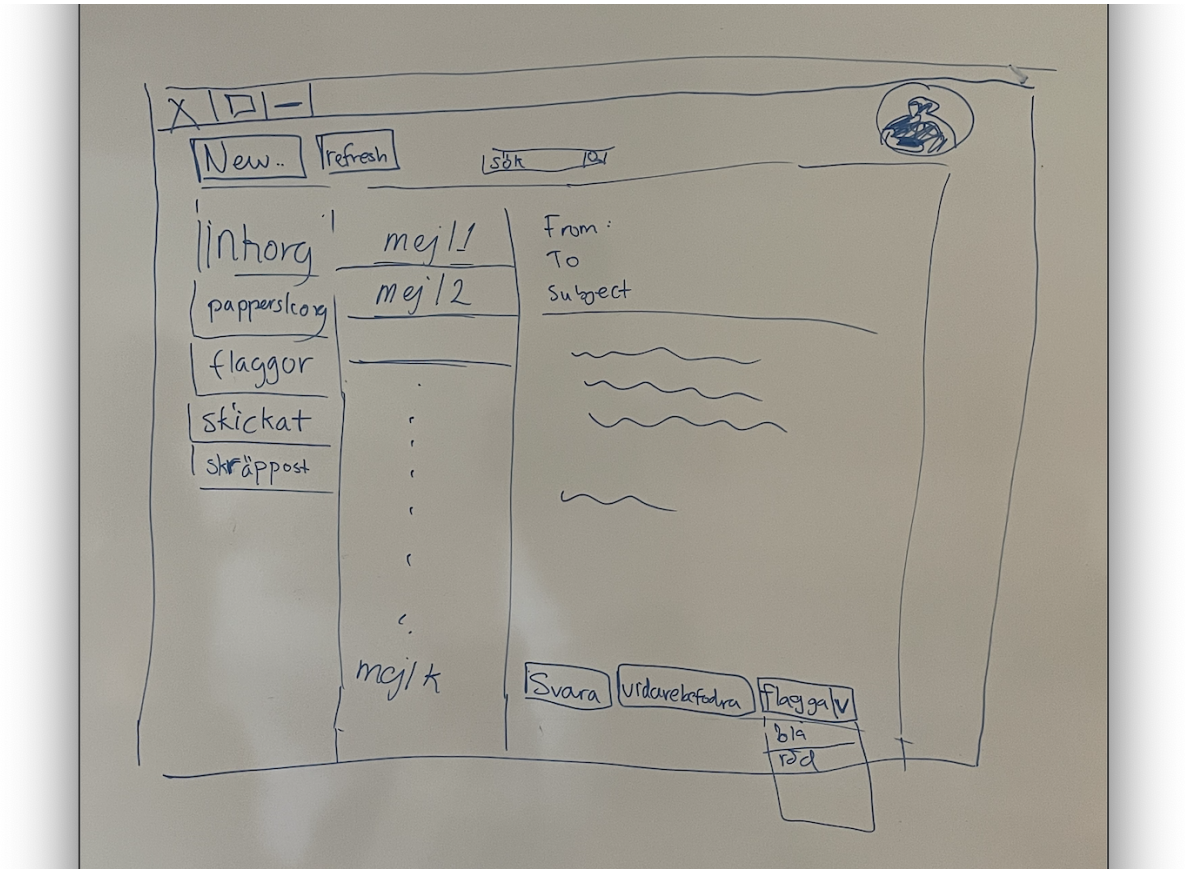
- The code should have been tested via unit tests.
- The code should have been tested by running the application and making sure everything runs.
- The code should be under the version control system git.
- The code should be thoroughly commented, including javadoc for all method signatures and classes.
- All javadoc comments should state the author of the class/method.
- The code should contain clear method and variable names.
- The Application should be able to run on Linux, OSX, and Windows.

2.3 User interface

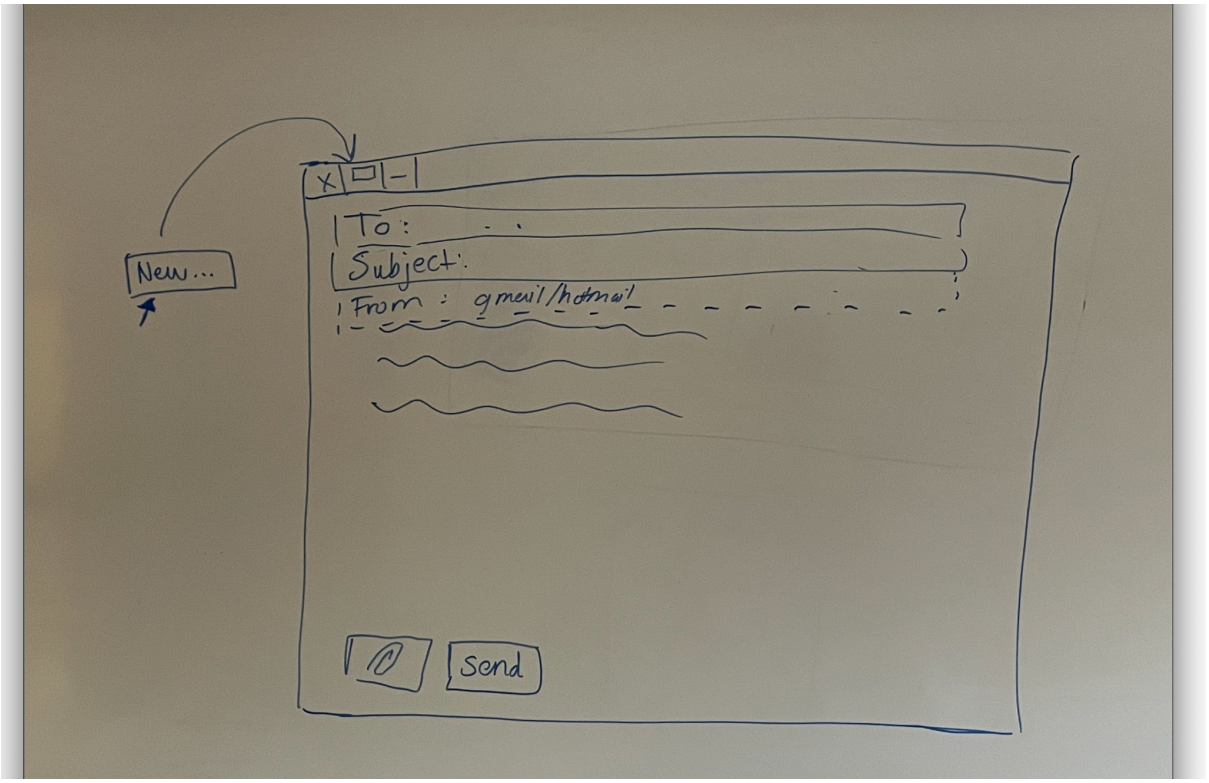
2.3.1 First sketches and drawings for our application interface

When we started planning the design for our application GUI in the first week, we started with some drawings/sketches of how we wanted the interface to look.

If we compare these sketches with the interface we ended up with, there are some small differences in component placements but overall the design is quite similar.



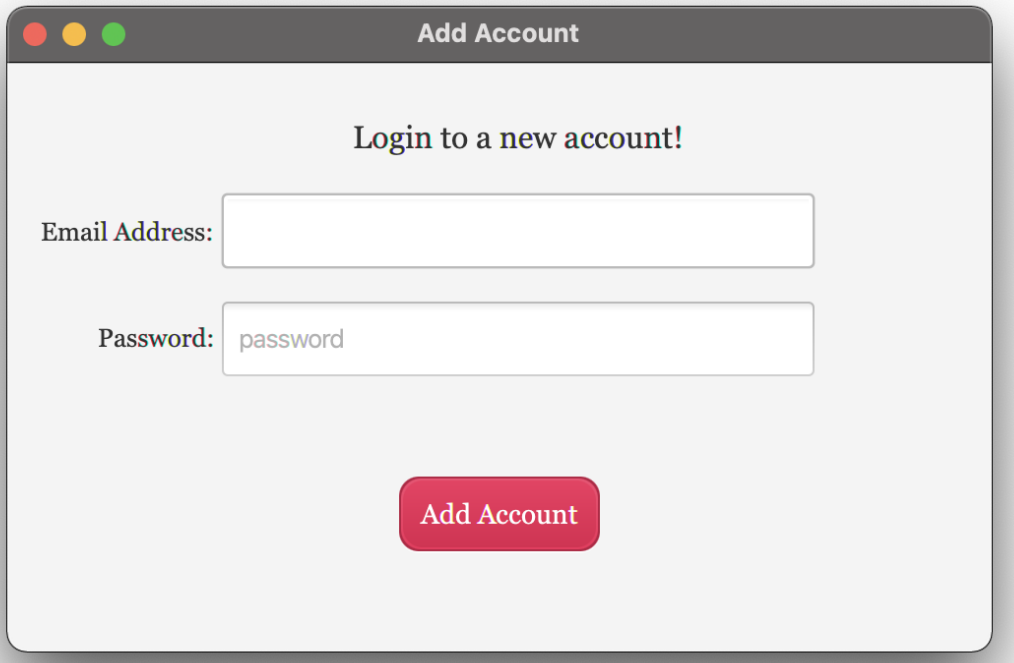
Sketch of the main view from week 1



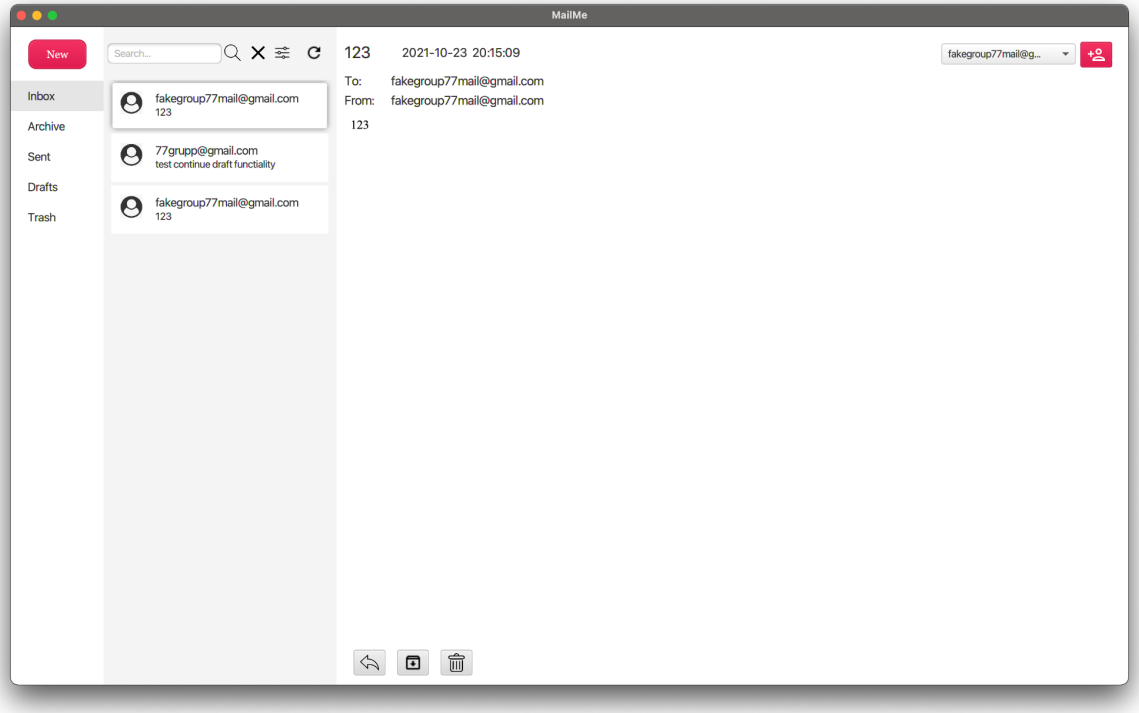
Sketch of the writing view from week 1

2.3.2 The application interface

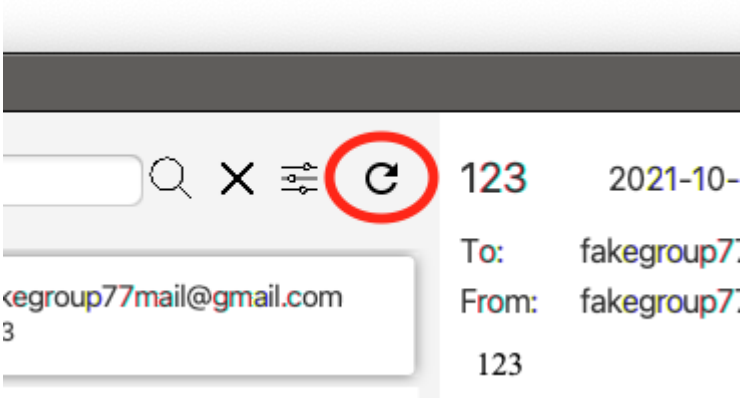
When starting the application for the first time, or when no accounts has yet to be added, This start view will be opened:



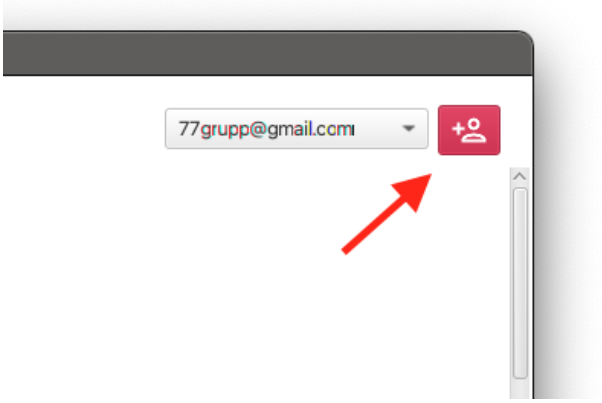
Here you can add a new account with an existing email address. When added, you will be directed to the main view when clicking on Add Account button:



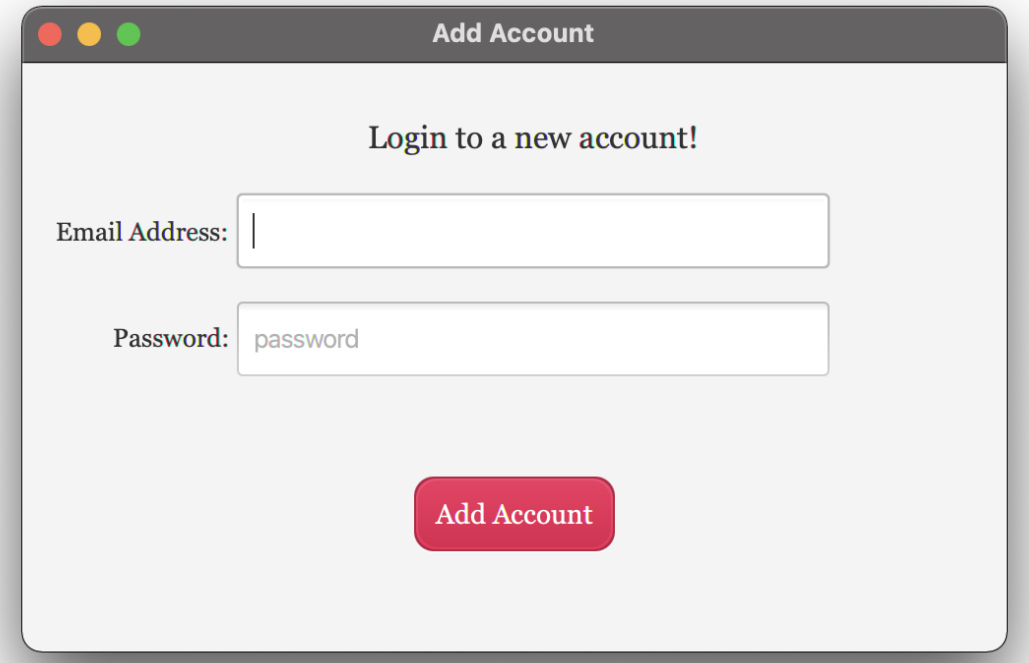
Here you can navigate through all the different mailboxes to the left and all the emails contained in them in the middle part. You can refresh to download new emails if you click on the refresh button next to the search bar:



If you want to add another account, you can click on the icon on the top right and click on “Add New Account”:

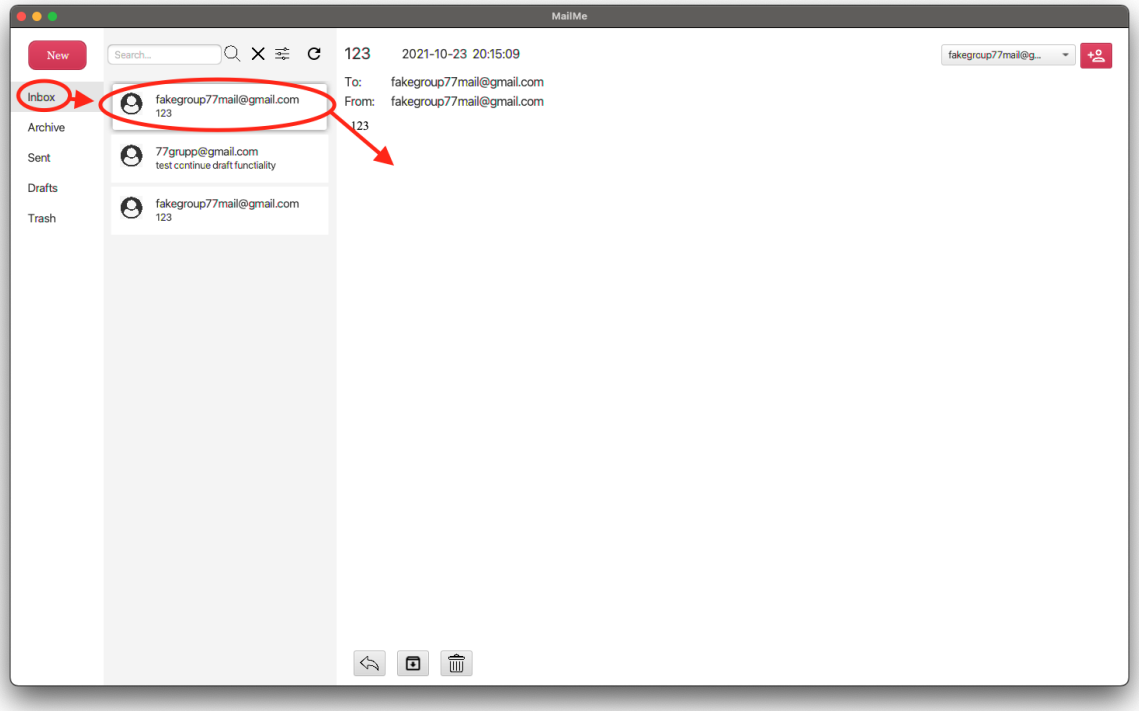


and a new window will pop up:

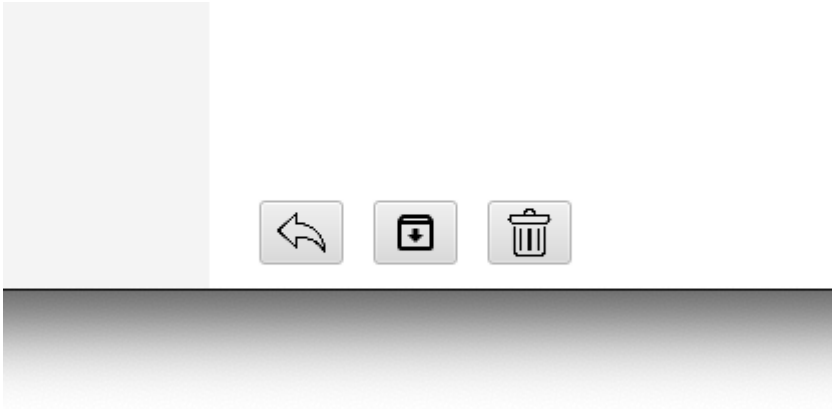


Here you can add your credentials for other email accounts.

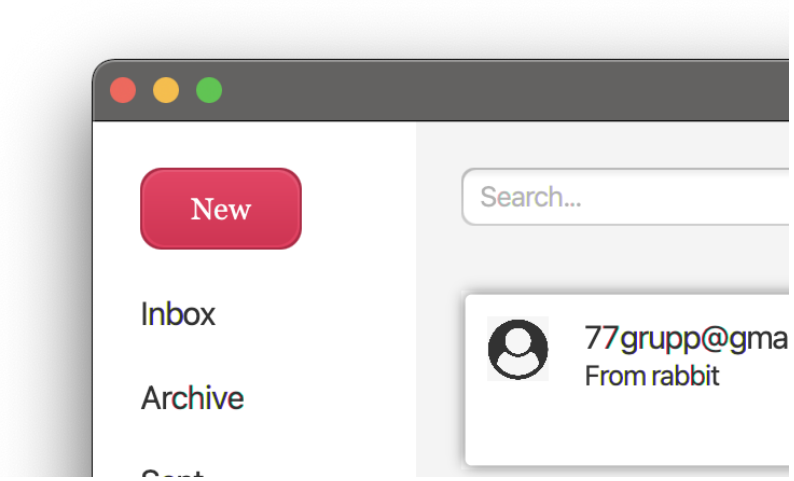
If you want to read an email, you click on the inbox and on the email you want to read, now you will see a reading view:



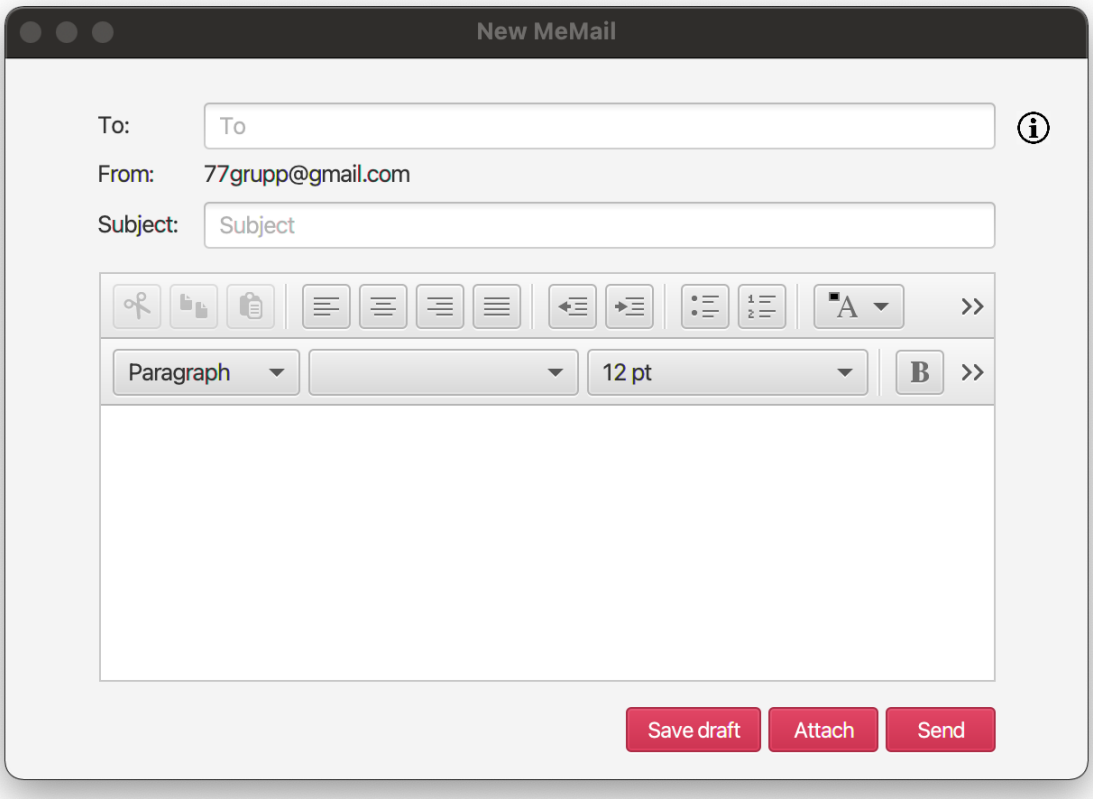
You can reply to the email, move it to trash or archive it by clicking on one of these buttons under the view:



When you click on the new button in the top left corner, a writing view will appear (the same view that appears when clicking on the reply button):

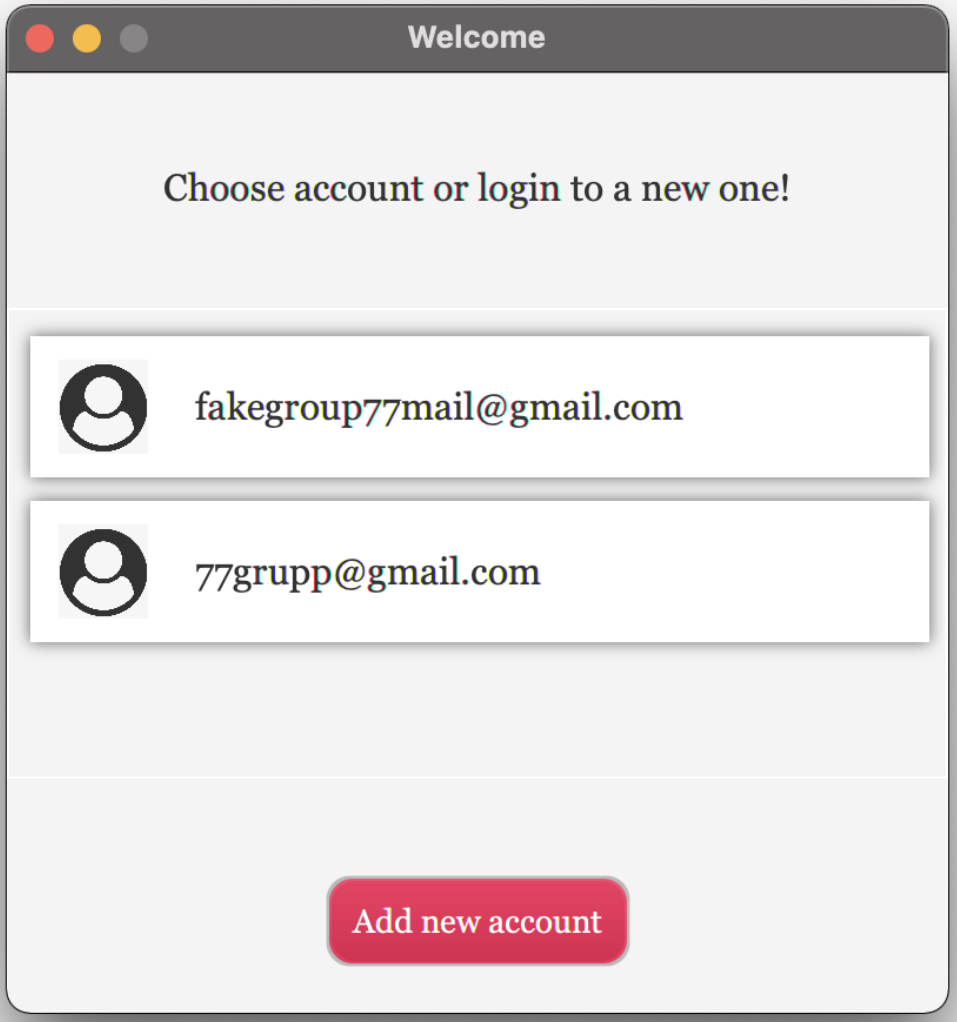


When you click on the button, this writing view appears:

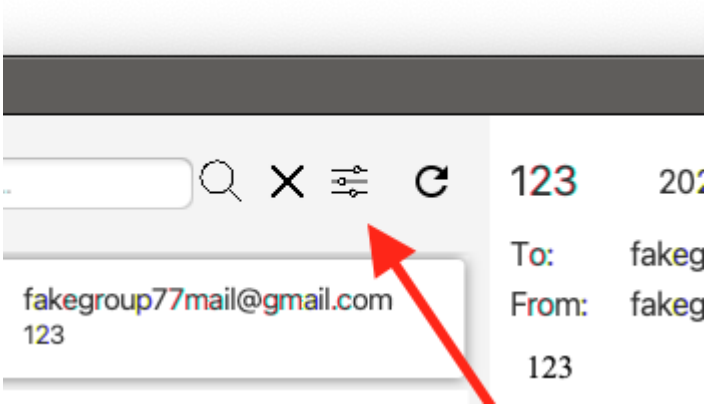


Here you can write new emails, with attachments if needed, and send them away.

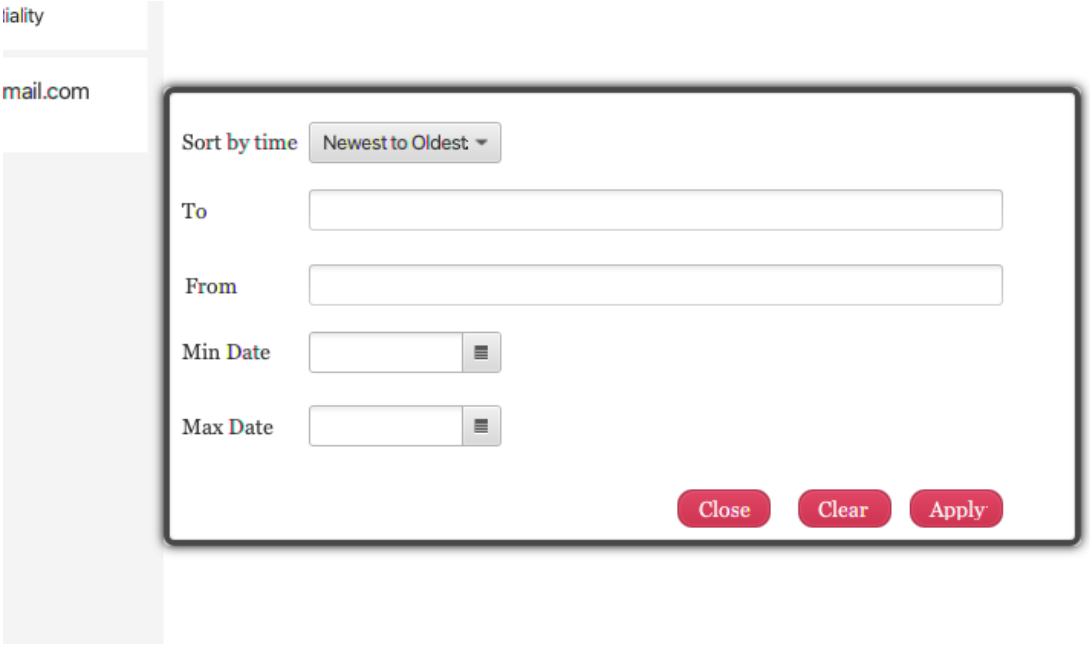
If two or more accounts has been added, and when reopening the application, this choice-screen will be opened, where you can choose which account you want login with(or add a new account):



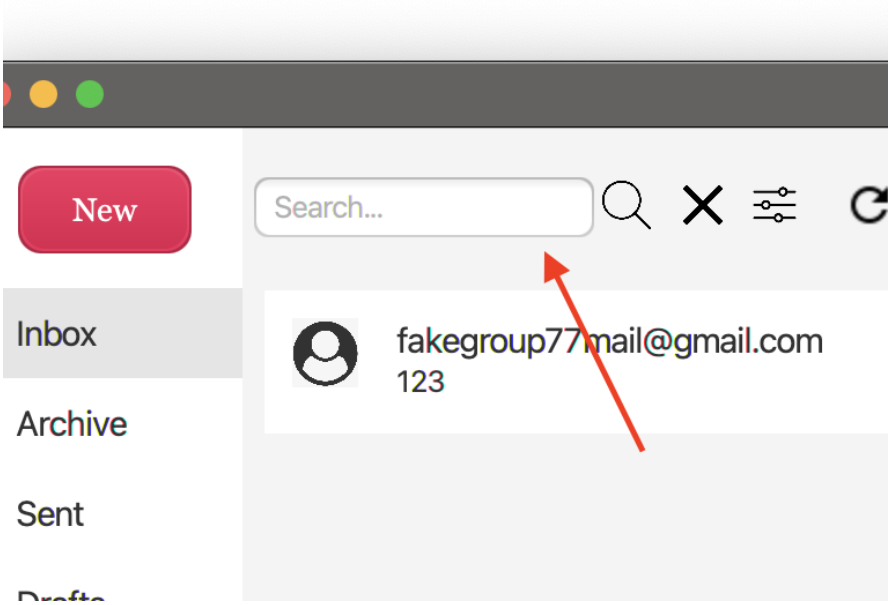
You can also sort emails in all of your different folders by clicking on the filter button next to the search bar:



This window will then pop up. Here you can choose to either sort by time (Newest to oldest/Oldest to newest), to a specific email address, from a specific email address or all the emails in a specific time window:

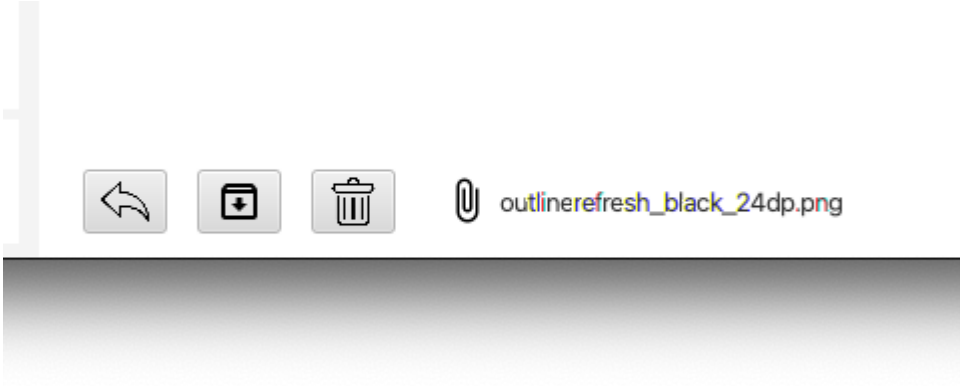


You can also search through your emails by entering keywords in the search bar found over the list of emails:

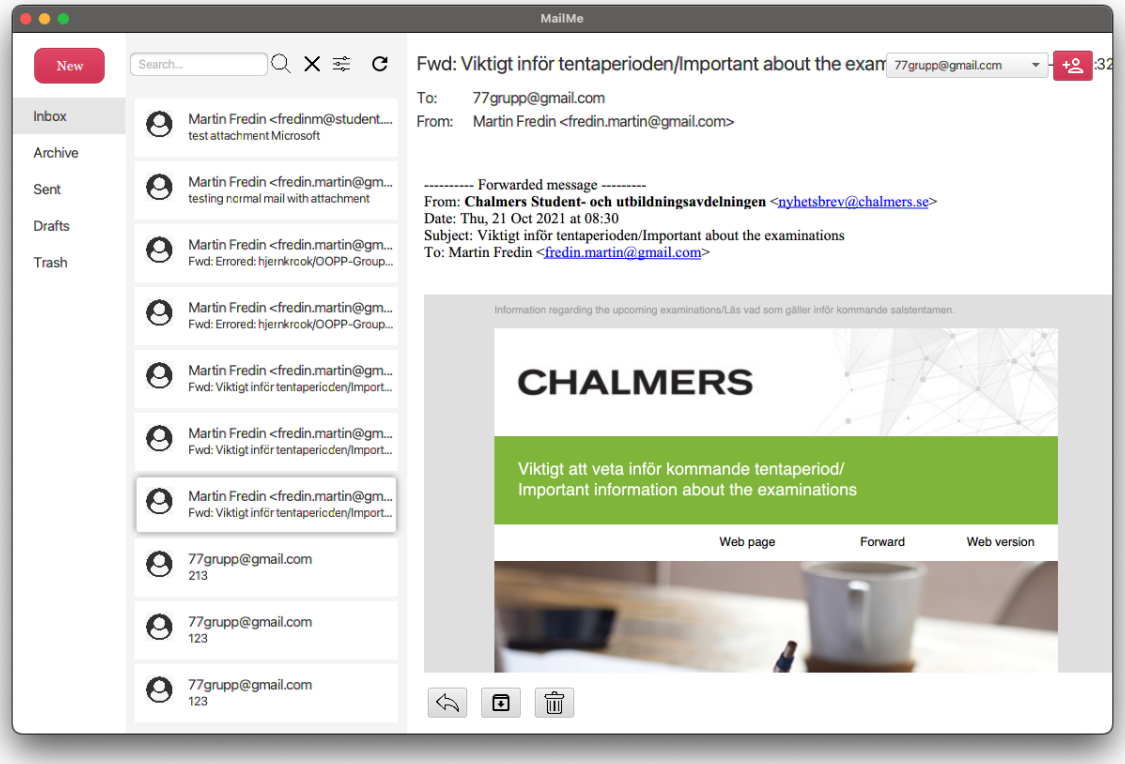


Properties of emails:

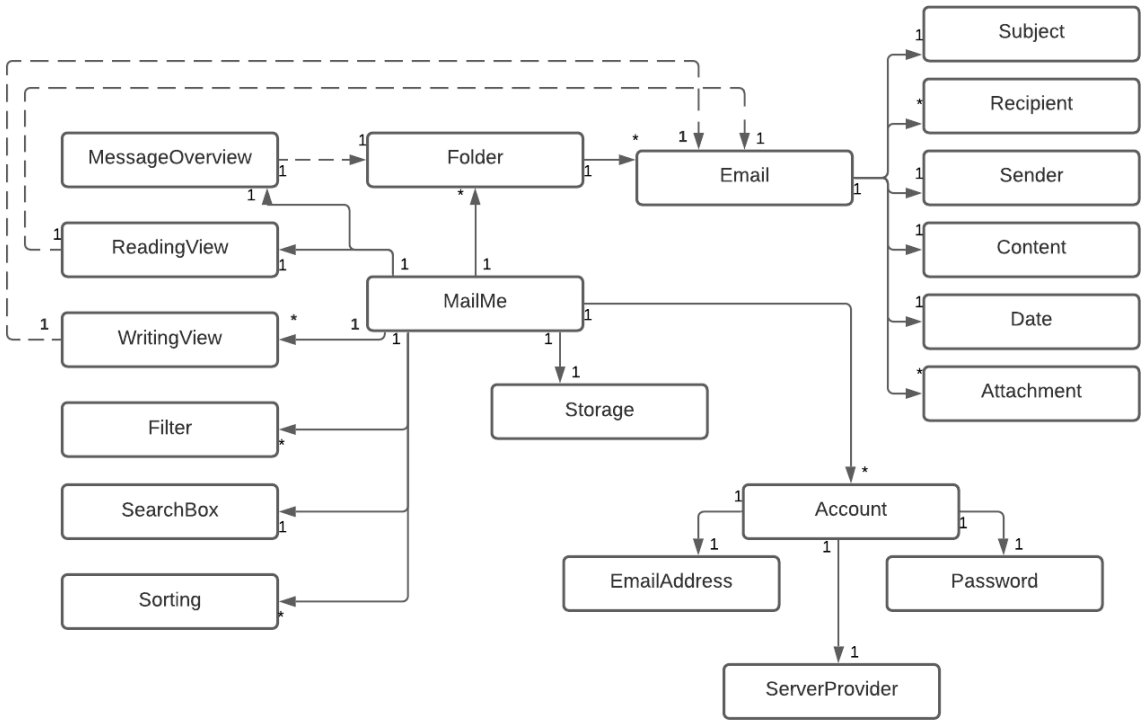
- You can send attachments with emails that will show up here:



And another property is that you can read and interact with HTML content within emails. here is an example:


MailMe
New
Search...
Inbox
Archive
Sent
Drafts
Trash
Martin Fredin <fredinm@student....
test attachment Microsoft
Martin Fredin <fredin.martin@gm...
testing normal mail with attachment
Martin Fredin <fredin.martin@gm...
Fwd: Errored: hjernkrook/OOPP-Group...
Martin Fredin <fredin.martin@gm...
Fwd: Errored: hjernkrook/OOPP-Group...
Martin Fredin <fredin.martin@gm...
Fwd: Viktigt inför tentaperioden/Import...
Martin Fredin <fredin.martin@gm...
Fwd: Viktigt inför tentaperioden/Import...
Martin Fredin <fredin.martin@gm...
Fwd: Viktigt inför tentaperioden/Import...
77grupp@gmail.com
213
77grupp@gmail.com
123
77grupp@gmail.com
123
Fwd: Viktigt inför tentaperioden/Important about the exam
77grupp@gmail.com
To: 77grupp@gmail.com
From: Martin Fredin <fredin.martin@gmail.com>
---------- Forwarded message ---------
From: Chalmers Student- och utbildningsavdelningen <nyhetsbrev@chalmers.se>
Date: Thu, 21 Oct 2021 at 08:30
Subject: Viktigt inför tentaperioden/Important about the examinations
To: Martin Fredin <fredin.martin@gmail.com>
Information regarding the upcoming examinations/Läs vad som gäller inför kommande salstentamen.
CHALMERS
Viktigt att veta inför kommande tentaperiod/
Important information about the examinations
Web page
Forward
Web version

3 Domain model



3.1 Class responsibilities

- MailMe: This represents the root of the application. This class coordinates and delegates between the different components of the application.
- Storage: This class is responsible for storing application-specific data, such as accounts and emails.
- Account: This class represents the abstract notion of an account, consisting of subparts:
    - EmailAddress: Represents an email address.
    - Password: Represents a password for a given account.
    - ServerProvider: The Email Service Provider that the account is linked to (where the account exists and gets its email service from).
- Email: Represent the abstract notion of an email, consisting of subparts:
    - Subject: Represents the subject of the email.
    - Recipient: Represents the recipient of the email (found in the ‘To’-field).
    - Sender: Represents the sender of the email (found in the From-field).
    - Content: Represents the text content of the email.
    - Date: Represents the sending or receiving date of the email.
    - Attachment: Represents any eventual files that were sent with the email.
- Folder: Represents a specific folder containing emails, such as ‘Inbox’, ‘Sent’ or

'Drafts'.
- MessageOverview: Shows all emails within a certain folder in a list-like overview.
- ReadingView: Displays the contents of an email.
- WritingView: Allows users to compose their own email.
- Filter: Allows users to filter emails in the current folder based on some filter condition, for example based on some specific date interval.
- SearchBox: Allows users to search for specific emails within the current folder that contain a specific search word.
- Sorting: Allows users to sort the emails according to a specific total ordering.

4 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)

Post Office Protocol. Available at: https://en.wikipedia.org/wiki/Post_Office_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)

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

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

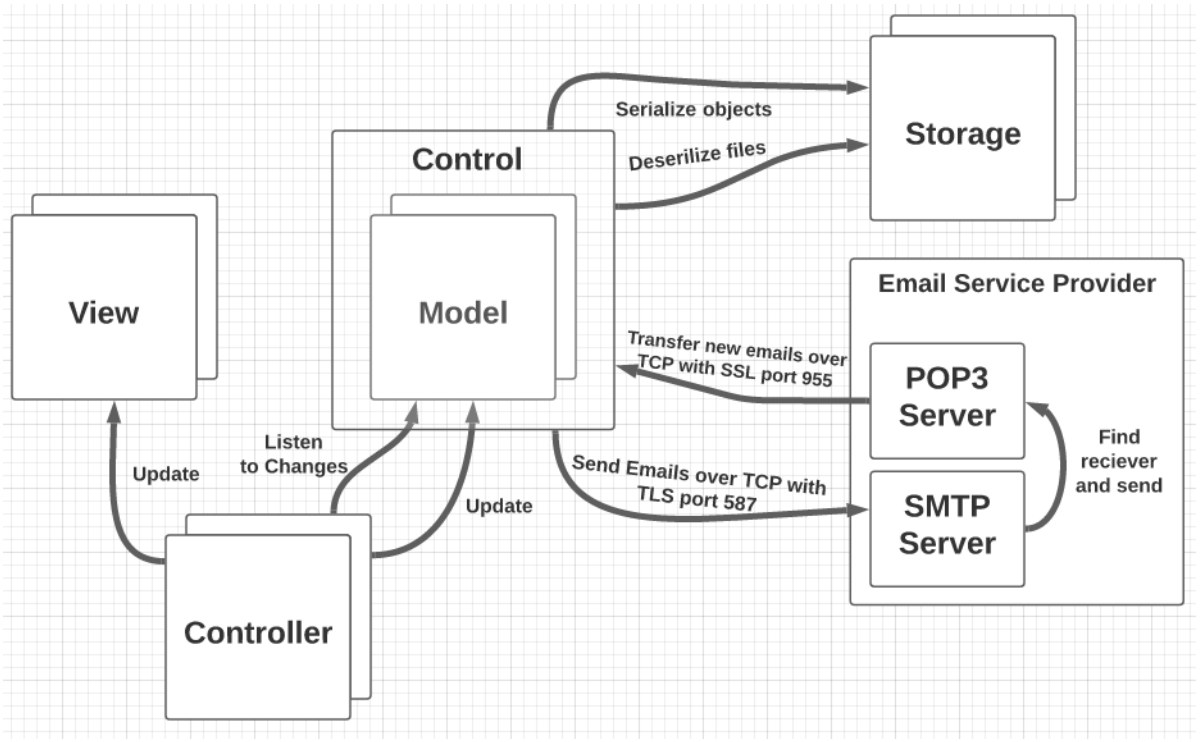


Figure 1. High level design model

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

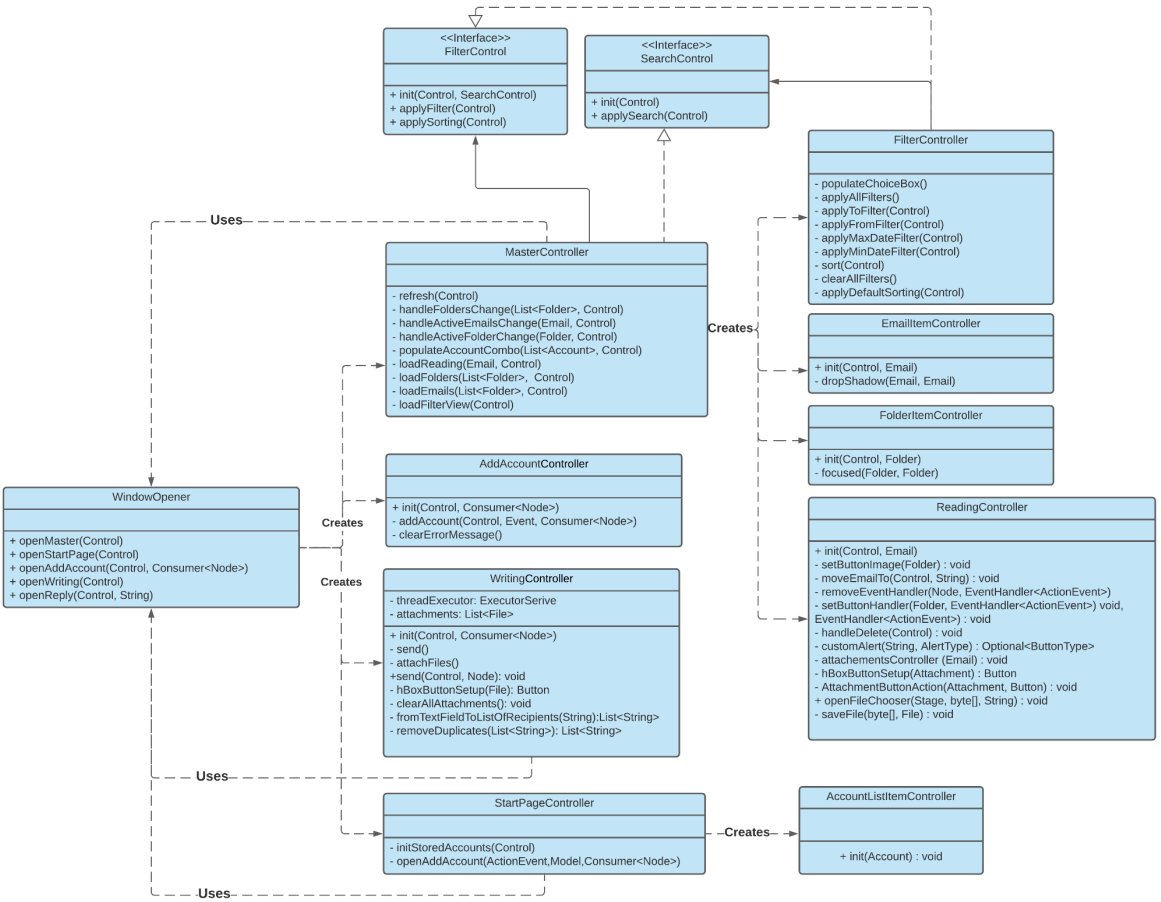


Figure 2. UML diagram of the controller package

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

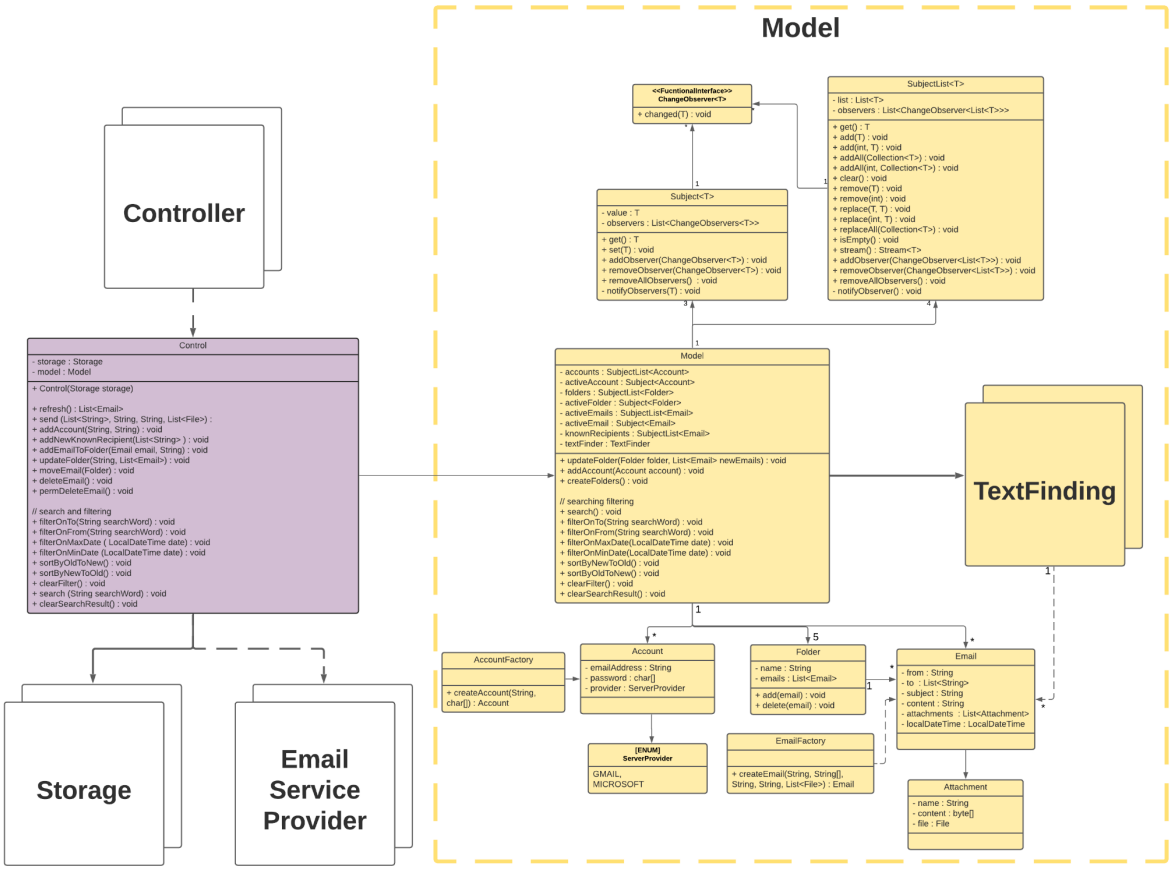


Figure 3. Package Model with relations to other packages.

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

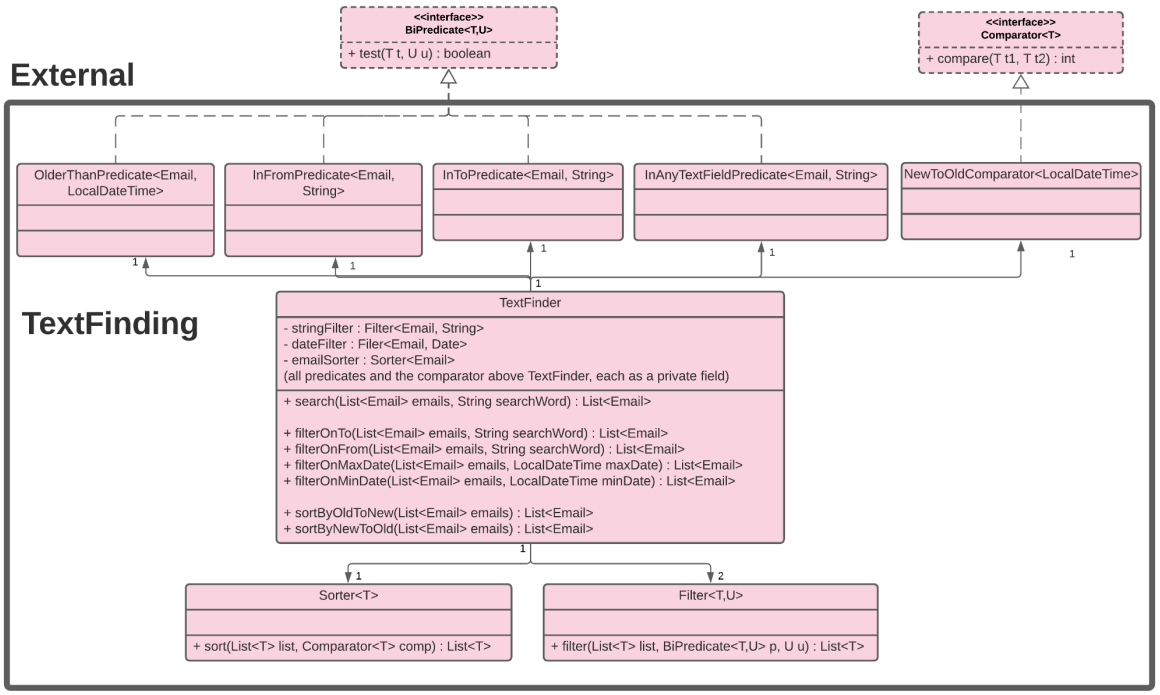


Figure 4.

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.

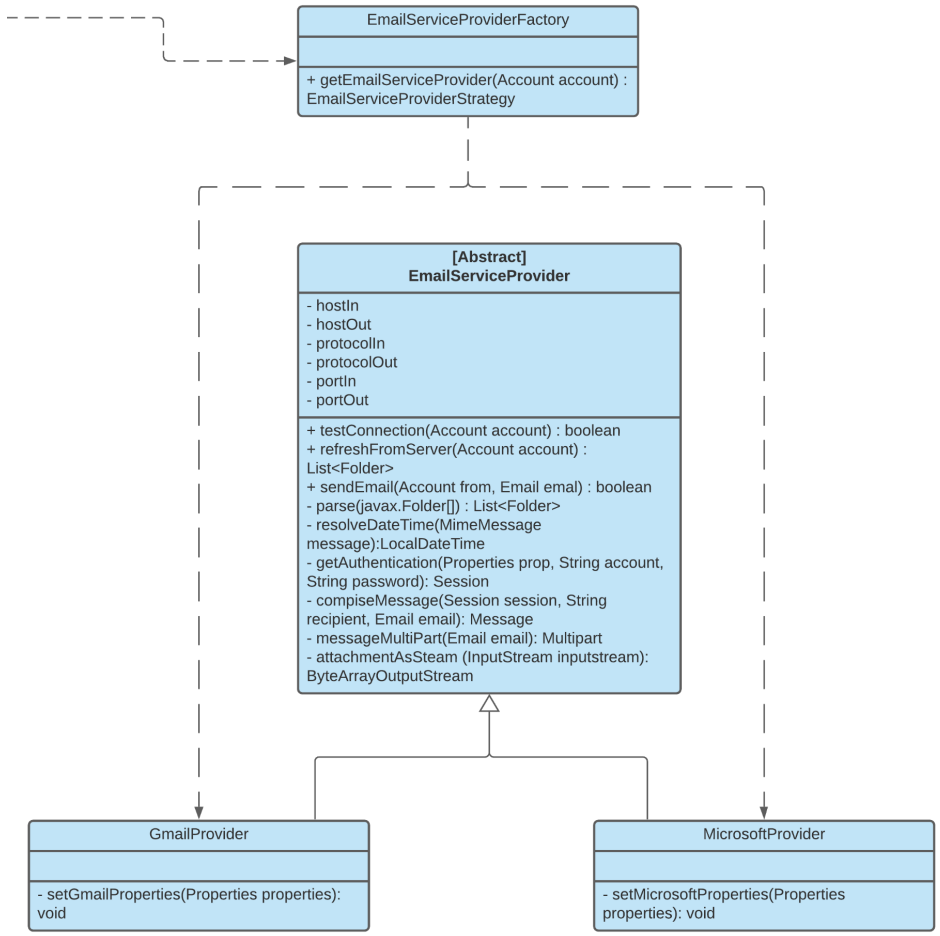


Figure 5.

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.

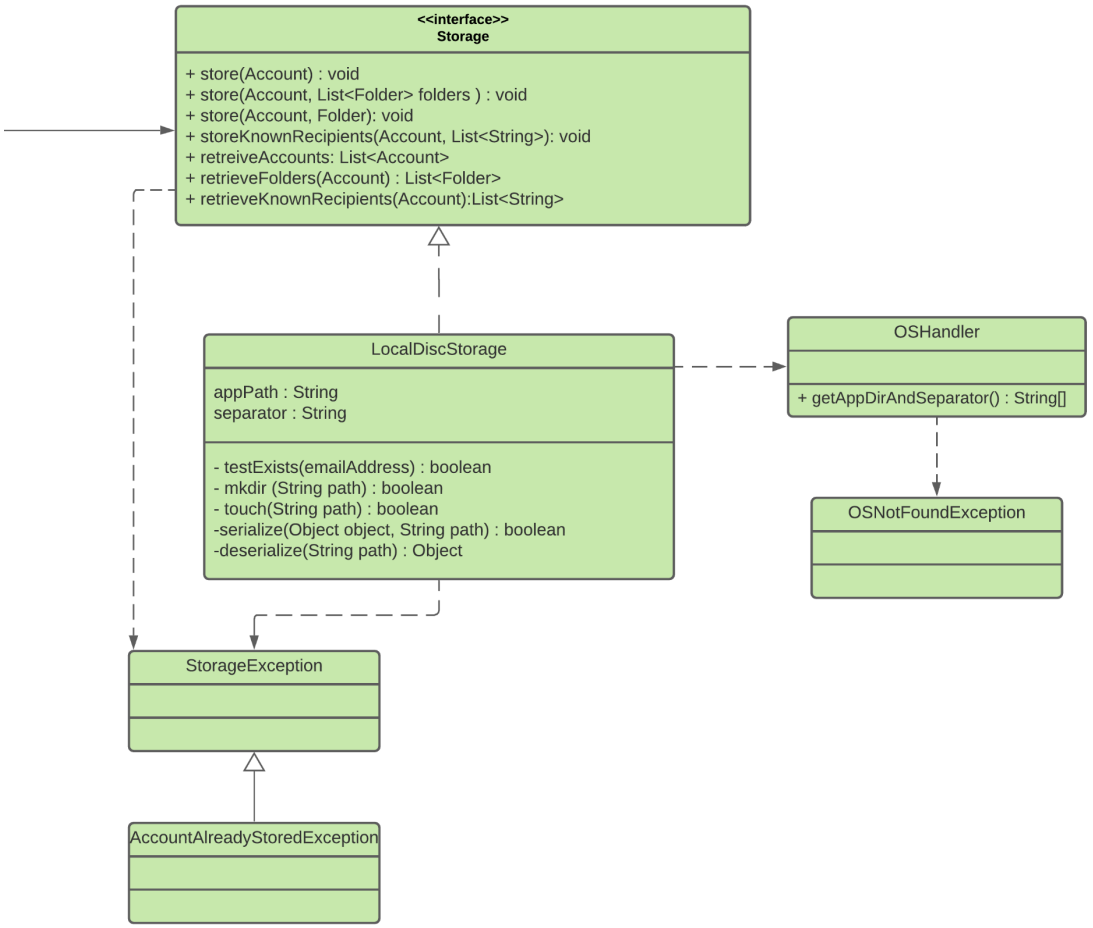


Figure 6.

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).

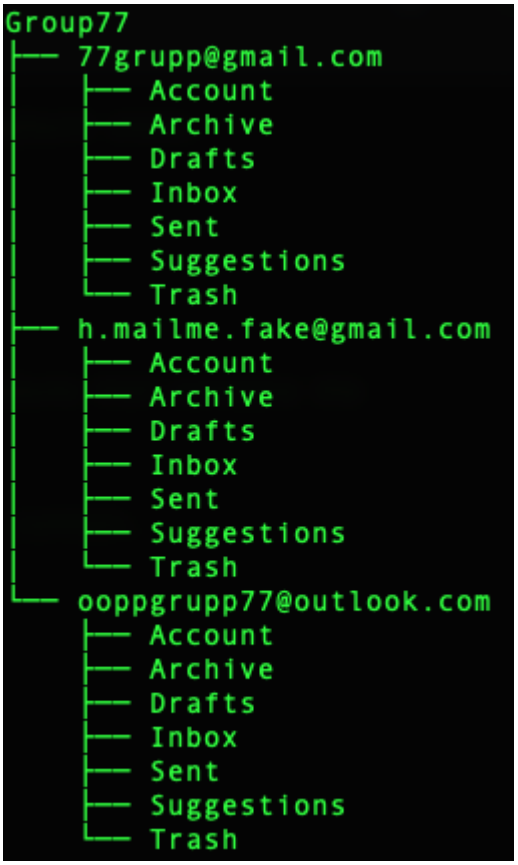


Figure 7. Directory structure of the application directory. Example with three added accounts.

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:

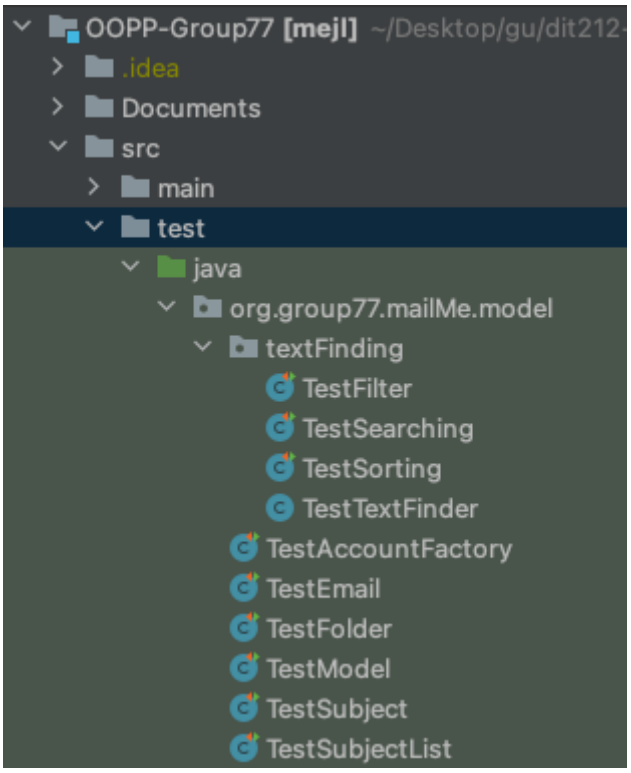


Figure 8. All junit test classes are located within the test folder. Test classes are divided by the package they belong to and contain tests for classes within those packages.

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

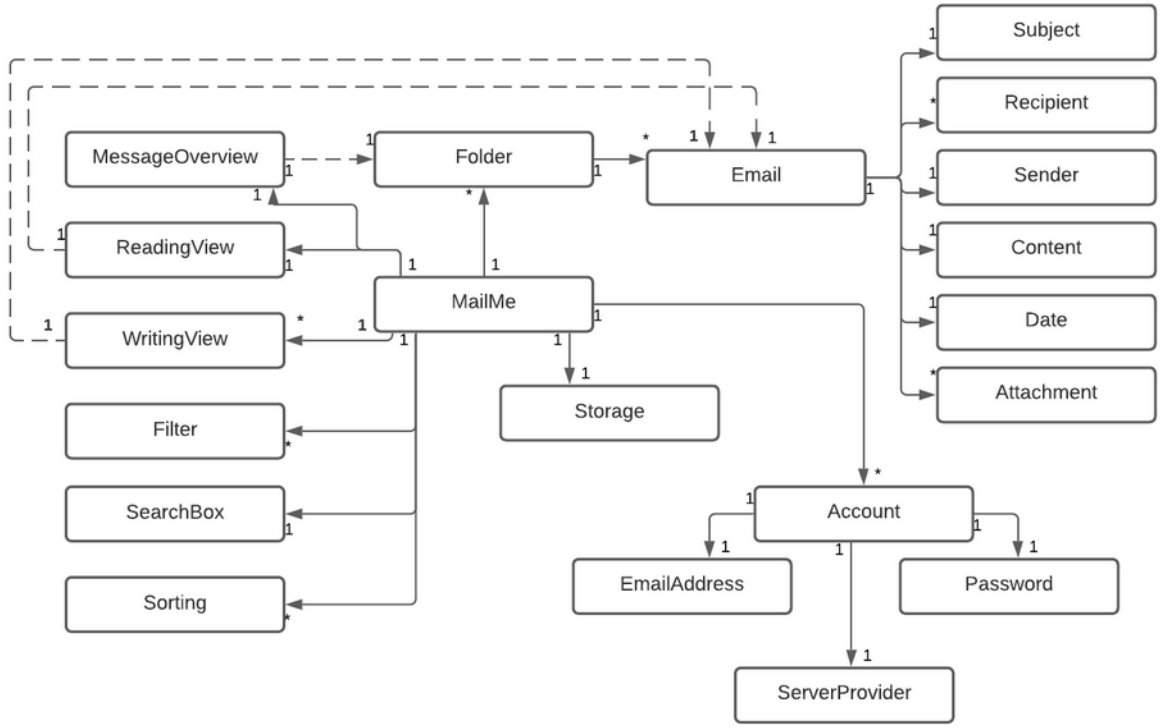


Figure 9. Domain Model

Peer review from OOPP-Group77 to TopDownBois

Our first impression of the game was that it is a fun idea and the animations look cool. You get an idea of what the game is about by just looking at the class names of the project. Also, the code uses a consistent coding style. Camel-case is used throughout all of the code.

Singleton pattern is used for View, ZombieObserver and MovableSubject. Since the use of singleton pattern makes it harder to test and control who has access to the instance, maybe consider using dependency injection instead.

The project uses a factory pattern for creating zombies. However, we question whether the ZombieFactory really follows the Single Responsibility Principle. In our opinion, factories should only create and return new instances of some class, not maintain any application state as is done in ZombieFactory. This could be located elsewhere.

The code contains javadoc comments for all methods which is good, but lacks comments that would have been a good addition elsewhere. For example, to make classes easier to understand and analyze, javadoc comments should be added describing the class' responsibility. Also, some additional comments within the method bodies would make the code easier to grasp. As a final note, the developers may want to add @author-tags to their javadoc comments, to give proper credit and also know who to ask about any given code snippet.

The naming of some classes and interfaces are confusing. The class TiledTestTwo is an example of this. The name indicates that the class should be some sort of test class, however it seems to be some sort of main class with a lot of responsibilities. Another example is the Zombies interface. The java naming conventions says that an interface should preferably be an adjective if the interface describes a behaviour. An interface can also be a noun for example List in the java collection library. A better name for Zombies would be for example PlayerPositionObserver because that is the functionality of the only method playerLocation(). This poor naming is very apparent when looking at the class signature for Coin.

```
public class Coin extends Sprite implements Movable, Zombies{
```

A much better interface-naming is the movable interface, which is very appropriate. Lastly, the ZombieObserver is wrongly named because the ZombieObserver is actually a ZombieObservable (which is why it contains a list of observers). It should probably be named ZombieSubject to keep consistent naming with the other observable MovableSubject.

The source code contains one package – *weapons* – but is not split into packages besides this. We would recommend creating a few distinct packages that groups classes with common responsibilities to make the code more structured and potentially easier to reuse, and also easier to understand. This would also make unnecessary dependencies (dependencies out from the package) more clear.

There is an attempt at an MVC pattern, however it is wrongly implemented because there are several instances where the model knows about the view. An example is Projectile which accesses the view and adds itself to the list of sprites. One of the main principles of MVC is that the Model should be isolated, which is clearly violated. One improvement to decouple view from model would be to move Sprite from the model to View. View would be responsible for attaching appropriate sprites for the objects in the model. Also consider dividing the classes into packages (model, view and controller) to make it easier to understand what classes belong to which part of the MVC.

One dependency which we question is the one from Player to PlayerController. It does not seem right that the model (Player) should have a controller as a field, as it makes the backend very coupled to the frontend and hinders reusability of the code.

The class RemindTask inside of Player could be private.

The abstract class Firearm is a good abstraction. It is properly named and has a good set of fields and methods. This makes it easy to add new weapons, however consider using dependency injection instead of creating the firearm inside of the constructor of the Player class. The application has multiple bad abstractions, mostly due to its overuse of class inheritance. For example, it is logical to assume that a Coin **is not a** Sprite. It would be much better to favor composition so Coin would instead **have a** Sprite (if Coin at all should have a Sprite since it should not know of the view).

Using composition will make the application much easier to maintain and extend because an object isn't constrained by a too narrow interface. Lastly, try to always use a broader static type than the actual dynamic type. This is in adherence to the Interface Segregation Principle. An easy fix would be to always use the less specific List type instead of ArrayList when possible.

```
private ArrayList<Zombie> zombies;
private ArrayList<Spawnpoint> spawnpoints = new ArrayList<>();
```

As far as we can see, there are no tests for the code. Tests should preferably be implemented before writing the code.

The project makes use of the observer pattern which is good to achieve a higher abstraction, decrease coupling and make the code more reusable. However, the current implementation of the observer pattern falls a bit short in reaching the proper abstraction level. In our opinion, the observers should implement some ObserverInterface and the observable implement some ObservableInterface. Then, perhaps the Main class could be responsible for adding observers to the observable's list of observers. Currently, observers such as Zombies do not implement any *onNotification()*- or *update()*-method called upon *notify()* from the observable. Instead, in CollisionController's *checkCollision()*-method, the observer's are called with the appropriate behaviour by checking their concrete class instance. This misses out on the opportunity of polymorphism that observer pattern is meant to provide, and increases coupling. Furthermore, observers such as Zombie now add themselves to the observable's list, instead of having the Main-class do it. This further increases coupling and risks reusability and maintainability.