System design document for

MailMe

authors:

Martin Fredin

Elin Hagman

David Zamanian

Alexey Ryabov

Hampus Jernkrook

date 2021-09-23

version 1.0

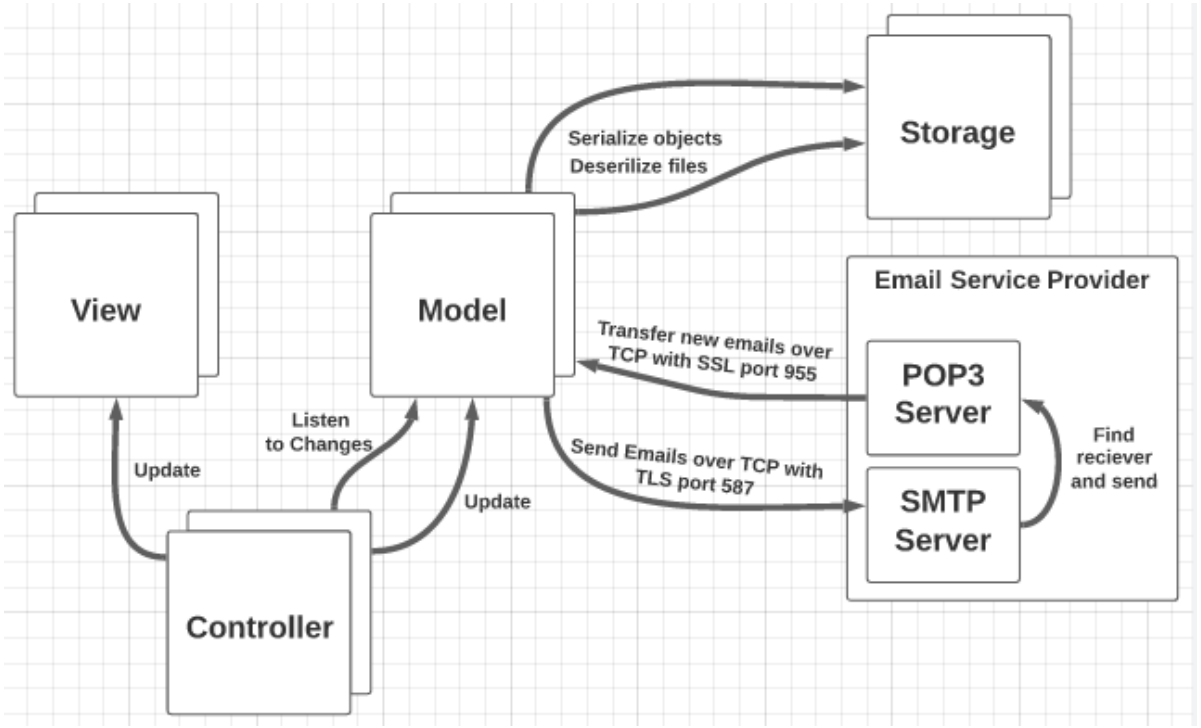
1 Introduction

MailMe is a simple email desktop application in which the user can login to already existing accounts from several different email providers. The main functions of the application are reading, writing and sending emails. There are some additional features which are commonly found on 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 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 from Serializing.
- **MVC (Model View Controller):** A design pattern where the model, view and controller are split up. (Model View Controller, 2021)

2 System architecture



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 state of the application at runtime, and to make requests to the services. 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 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.

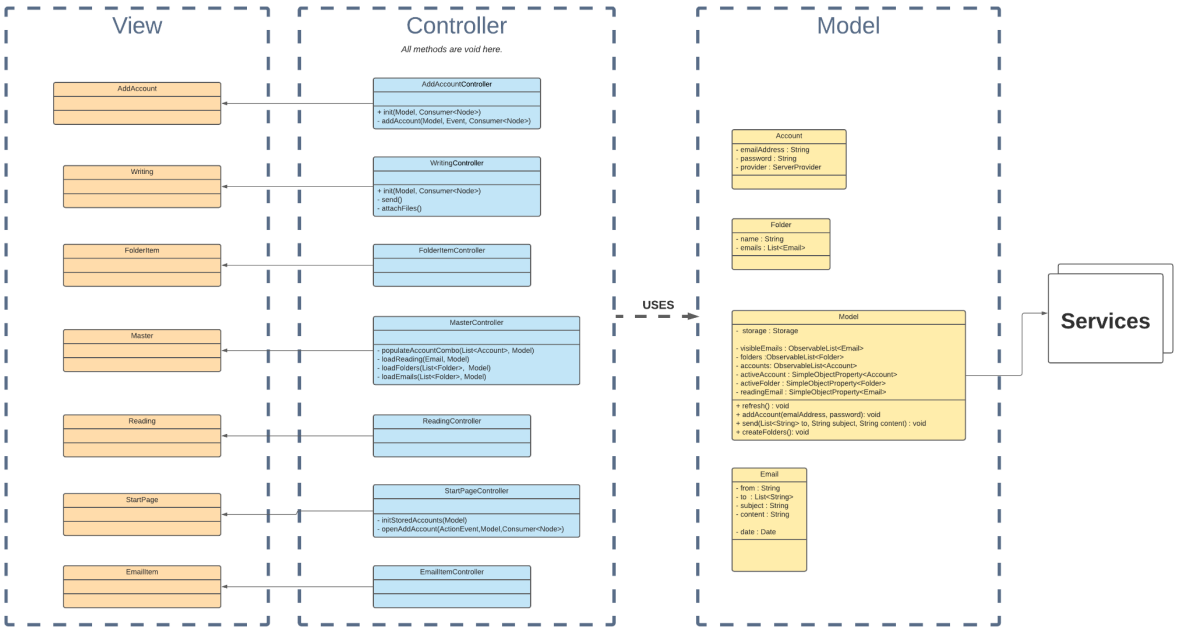
3 System design

The application consists of packages:

- View
- Controller
- Model
- Services, with subpackages:
    - Email Service Provider
    - Storage

All packages are illustrated with UML class diagrams in the Appendix.

On a high level, the design looks like this (see Appendix figure 1 for a more detailed image):



The MVC pattern is implemented using an observer pattern. Controller classes listen to specific parts of the model's state and perform actions accordingly to update the view.

Other design patterns that are used in the applications are:

- Aggregate pattern: This pattern is used for the filtering and sorting functionality of the application. Model (the class) knows not about all different classes working together to solve this task, but instead only knows of the TextFinder class, which works as the aggregate root. Behind the scenes, TextFinder uses many other classes to get the filtering and sorting done.

- Factory pattern: This pattern is used to create concrete Account instances, hiding from the Model class how the ServerProvider field of the Account instance is chosen. We also use the factory pattern for creating concrete EmailServiceProviders, again to hide from the Model class how the creation of instances is done.

- Template pattern for storage and concrete ESPs?

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 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. This is the only option for our solution due to the fact that we need to send the credentials in plain text to the ESP to be able to retrieve the emails and folders, which makes it impossible for us to hash the password.

TODO: maybe mention the directory structure?

5 Quality

Our application, by design, is divided into several packages/ modules: model, services, controller and view (fxml resources). Our controller and view packages are difficult to test with Junit. The view package is tested manually with a try and error approach by running the application. Currently, the Service package is also tested only via running the application and seeing that everything works, as testing these parts involve some difficulties. For example, testing storage requires working with the user’s local machine’s storage, which is different for different users. The model is tested with Junit. For each class in this package there exists a test class. The aim for coverage of the Model package was set to 90% or higher.

Junit tests are located in the test folder within the main folder of the OOPP-Group77 project, Figure 1.

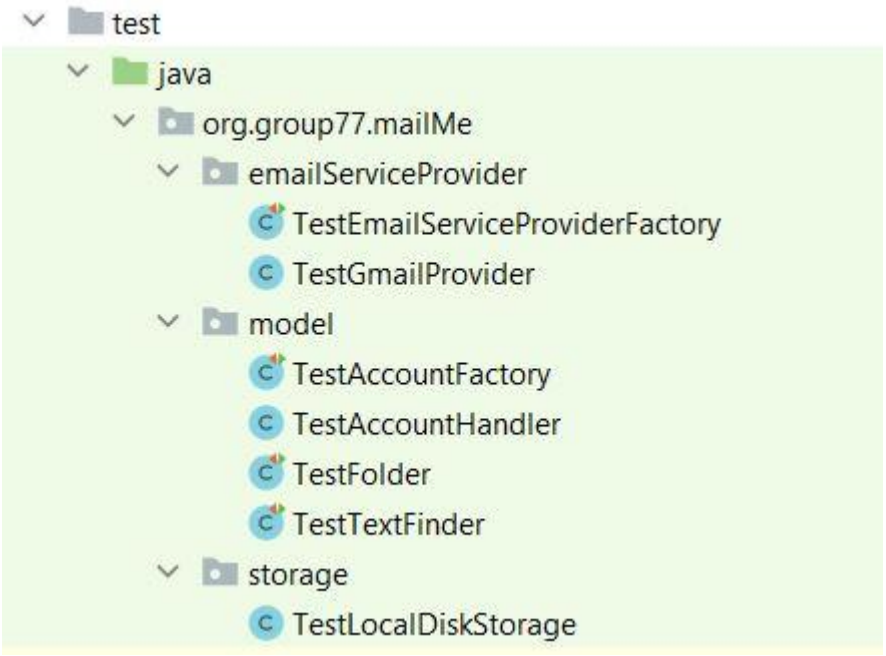


Figure 1: All junit test classes are located within the test folder. Test classes are divided by the package they belong to and contain tests for classes within those packages.

Travis CI link: https://app.travis-ci.com/github/hjernkrook/OOPP-Group77

List of all known issues:

1) Email from-date does not display correctly in the reading view.
2) Reading view: to- textfield display email address within the brackets.
3) Clicking and canceling add new account leaves the current account-text as ‘add new account.
4) Attachment is not shown with the email after having attached the file.
5) Adding a second account does not always update the active account shown correctly (sometimes it is blank).
6) Since the migration to a new branch due to a design change, PMD and JDepend no longer work 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. This account information is stored on the local harddrive. Depending on the operative system, the directory of the account information can vary.

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.

At this moment, only gmail support is available. Our intent is to include Microsoft Outlook support in the future.

For testing purposes multiple “fake” gmail accounts were created.

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)

7 Appendix

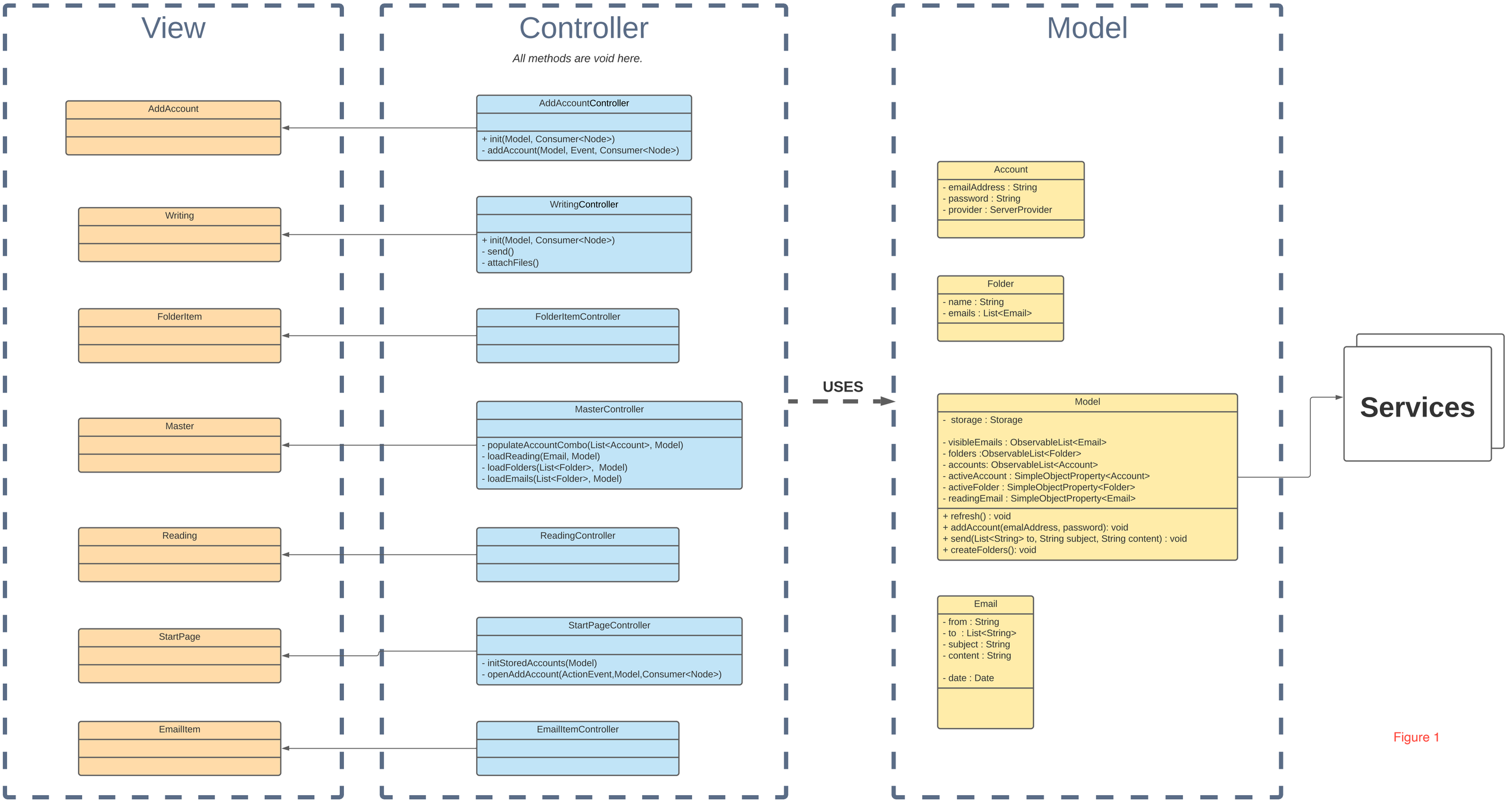

View
Controller
All methods are void here.
Model
AddAccount
AddAccountController
+ init(Model, Consumer<Node>)
- addAccount(Model, Event, Consumer<Node>)
Writing
WritingController
+ init(Model, Consumer<Node>)
- send()
- attachFiles()
FolderItem
FolderItemController
Master
MasterController
- populateAccountCombo(List<Account>, Model)
- loadReading(Email, Model)
- loadFolders(List<Folder>, Model)
- loadEmails(List<Folder>, Model)
Reading
ReadingController
StartPage
StartPageController
- initStoredAccounts(Model)
- openAddAccount(ActionEvent,Model,Consumer<Node>)
EmailItem
EmailItemController
USES
Account
- emailAddress : String
- password : String
- provider : ServerProvider
Folder
- name : String
- emails : List<Email>
Model
- storage : Storage
- visibleEmails : ObservableList<Email>
- folders :ObservableList<Folder>
- accounts: ObservableList<Account>
- activeAccount : SimpleObjectProperty<Account>
- activeFolder : SimpleObjectProperty<Folder>
- readingEmail : SimpleObjectProperty<Email>
+ refresh() : void
+ addAccount(emalAddress, password): void
+ send(List<String> to, String subject, String content) : void
+ createFolders(): void
Email
- from : String
- to : List<String>
- subject : String
- content : String
- date : Date
Services

Figure 1

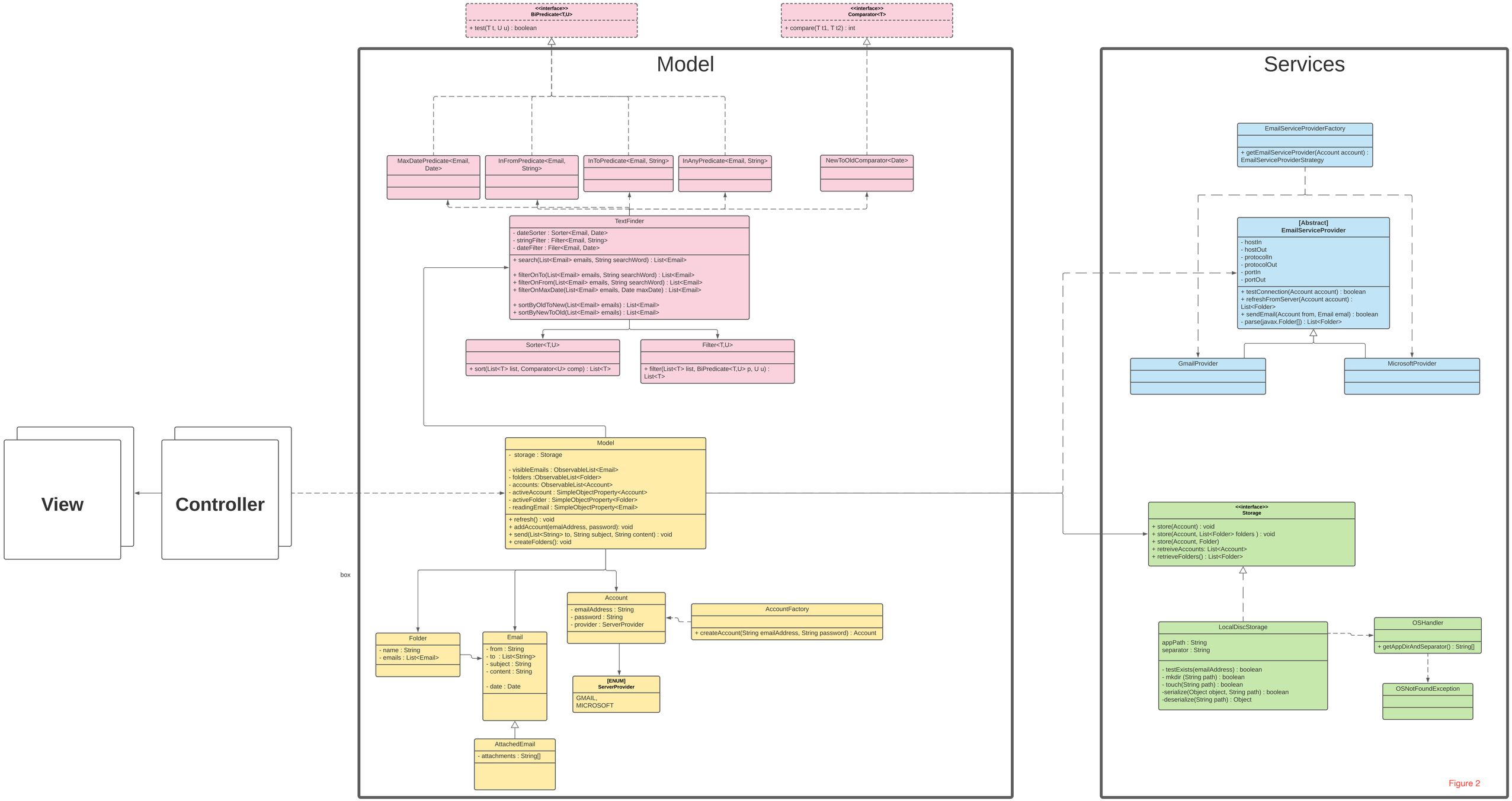

<<interface>>
BiPredicate<T,U>
+ test(T t, U u) : boolean
<<interface>>
Comparator<T>
+ compare(T t1, T t2) : int
Model
MaxDatePredicate<Email, Date>
InFromPredicate<Email, String>
InToPredicate<Email, String>
InAnyPredicate<Email, String>
NewToOldComparator<Date>
TextFinder
- dateSorter : Sorter<Email, Date>
- stringFilter : Filter<Email, String>
- dateFilter : Filer<Email, Date>
+ search(List<Email> emails, String searchWord) : List<Email>
+ filterOnTo(List<Email> emails, String searchWord) : List<Email>
+ filterOnFrom(List<Email> emails, String searchWord) : List<Email>
+ filterOnMaxDate(List<Email> emails, Date maxDate) : List<Email>
+ sortByOldToNew(List<Email> emails) : List<Email>
+ sortByNewToOld(List<Email> emails) : List<Email>
Sorter<T,U>
+ sort(List<T> list, Comparator<U> comp) : List<T>
Filter<T,U>
+ filter(List<T> list, BiPredicate<T,U> p, U u) : List<T>
Model
- storage : Storage
- visibleEmails : ObservableList<Email>
- folders :ObservableList<Folder>
- accounts: ObservableList<Account>
- activeAccount : SimpleObjectProperty<Account>
- activeFolder : SimpleObjectProperty<Folder>
- readingEmail : SimpleObjectProperty<Email>
+ refresh() : void
+ addAccount(emalAddress, password): void
+ send(List<String> to, String subject, String content) : void
+ createFolders(): void
View
Controller
box
Folder
- name : String
- emails : List<Email>
Email
- from : String
- to : List<String>
- subject : String
- content : String
- date : Date
AttachedEmail
- attachments : String[]
Account
- emailAddress : String
- password : String
- provider : ServerProvider
[ENUM]
ServerProvider
GMAIL,
MICROSOFT
AccountFactory
+ createAccount(String emailAddress, String password) : Account
Services
EmailServiceProviderFactory
+ getEmailServiceProvider(Account account) : EmailServiceProviderStrategy
[Abstract]
EmailServiceProvider
- hostIn
- hostOut
- protocolIn
- protocolOut
- portIn
- portOut
+ testConnection(Account account) : boolean
+ refreshFromServer(Account account) : List<Folder>
+ sendEmail(Account from, Email emal) : boolean
- parse(javax.Folder[]) : List<Folder>
GmailProvider
MicrosoftProvider
<<interface>>
Storage
+ store(Account) : void
+ store(Account, List<Folder> folders) : void
+ store(Account, Folder)
+ retreiveAccounts: List<Account>
+ retrieveFolders() : List<Folder>
LocalDiscStorage
appPath : String
separator : String
- testExists(emailAddress) : boolean
- mkdir (String path) : boolean
- touch(String path) : boolean
-serialize(Object object, String path) : boolean
-deserialize(String path) : Object
OSHandler
+ getAppDirAndSeparator() : String[]
OSNotFoundException

Figure 2