System design document for MailBowser

Version: 2.0

Date: 2015-05-31

Authors: Jesper Jaxing, Oscar Evertsson, Mats Högberg, Filip Hallqvist

1 Introduction

1.1 Design Goals

The design must be modular. The code base must be separated into distinct packages, and it must be possible to use most of the packages independently. The design must also be testable, and allow for easy swapping of individual components without affecting the rest of the system.

1.2 Definitions, acronyms and abbreviations

- GUI: Graphical User Interface. The face, or view, of the application.
- Java: A platform independent programming language.
- MVC: A design pattern where the application is divided into three distinct parts - model, view and controller. This is done to separate the view from the underlying logic.
- MVP: A modified version of the MVC pattern where the controller is replaced by a presenter.
- Markdown: A type of markup language used for formatting text. Markdown is compiled to regular HTML.
- Email: A method of exchanging digital messages from a sender to one or more recipients.
- Email message: A single message sent via email.

2 System design

2.1 Overview

The application uses a modified version of the MVP (Model-View-Presenter) pattern.

The MVP pattern is a derivation of the MVC pattern (Chandel, 2009), where you replace the regular controller with a *presenter*. The difference between the two patterns is that the view in MVP has much less responsibility than the one found in MVC. In MVP, the view is only responsible for showing data to the user - nothing more. The presenter listens to the model, and then tells the view what data to display. Apart from this, the two patterns are identical.

The modified version of MVP used in this application utilizes an event bus as a layer between the models and the presenters (see figure 1 and 2). The event bus is a singleton that functions as a central hub for all events. All presenters that are interested in changes of any kind registers to the event bus upon initialization. When something happens in the application, an event is sent to the event bus.The event bus then distributes the event to all registered presenters. The usage of an event bus greatly simplifies the process of connecting observers to observables, since the event bus is a singleton and therefore can be accessed easily by all models and presenters in the application.

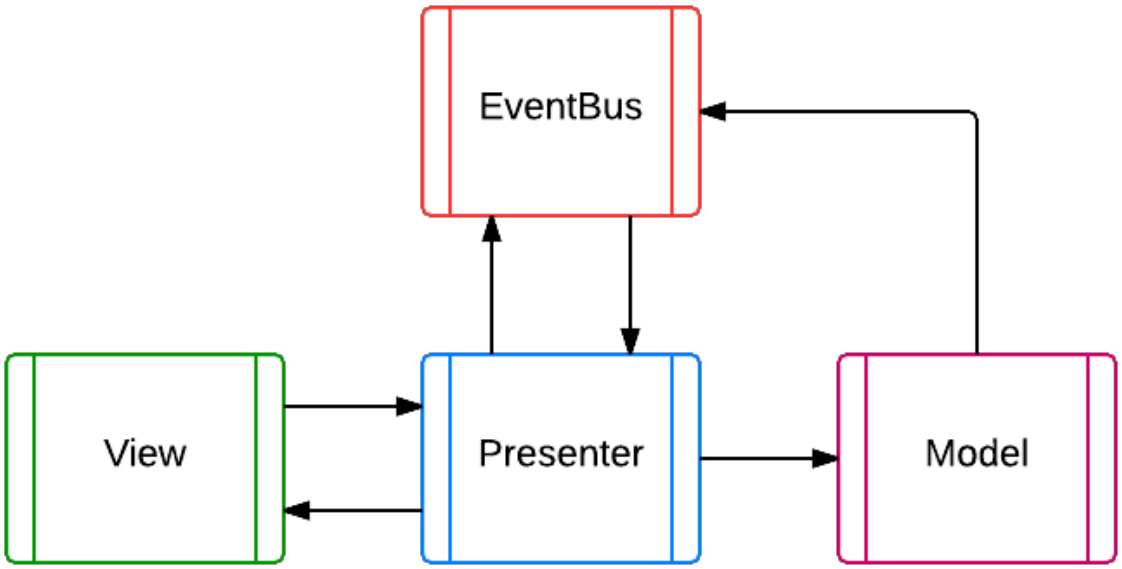


Figure 1: MVP pattern with event bus

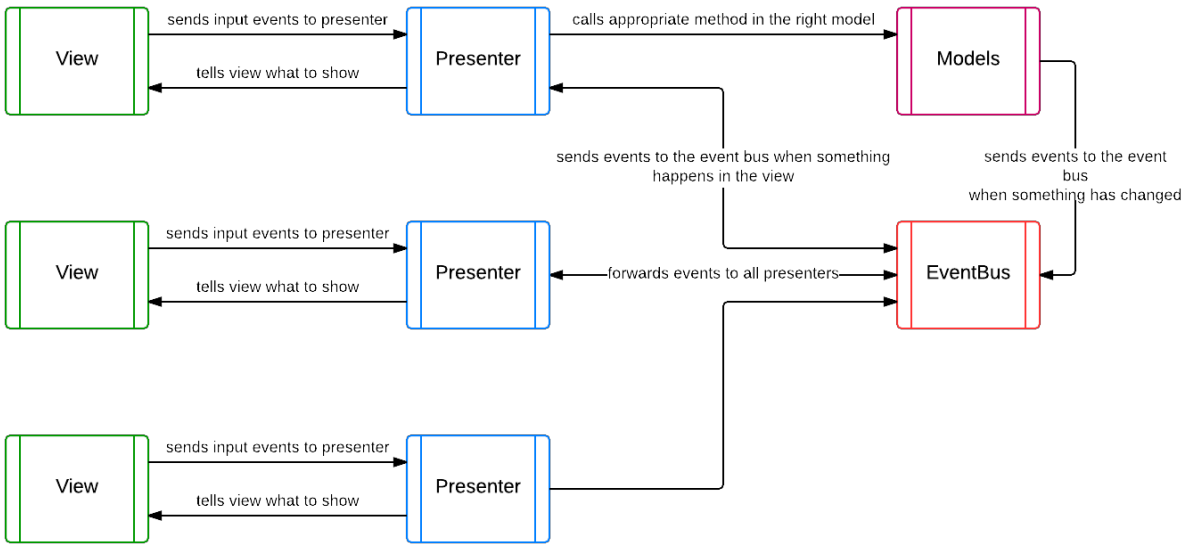


Figure 2: More detailed application flowchart

2.1.2 Graphical User Interface

The application has a clean GUI (see figure 3). The GUI is composed of views, and each view has it's own dedicated presenter. To keep the design modular, the views are divided into as many subviews as possible. The key philosophy here is that each view-presenter pair only should be responsible for one thing. This is in accordance to the design principle *Single Responsiblity Principle* (Martin, 2002).

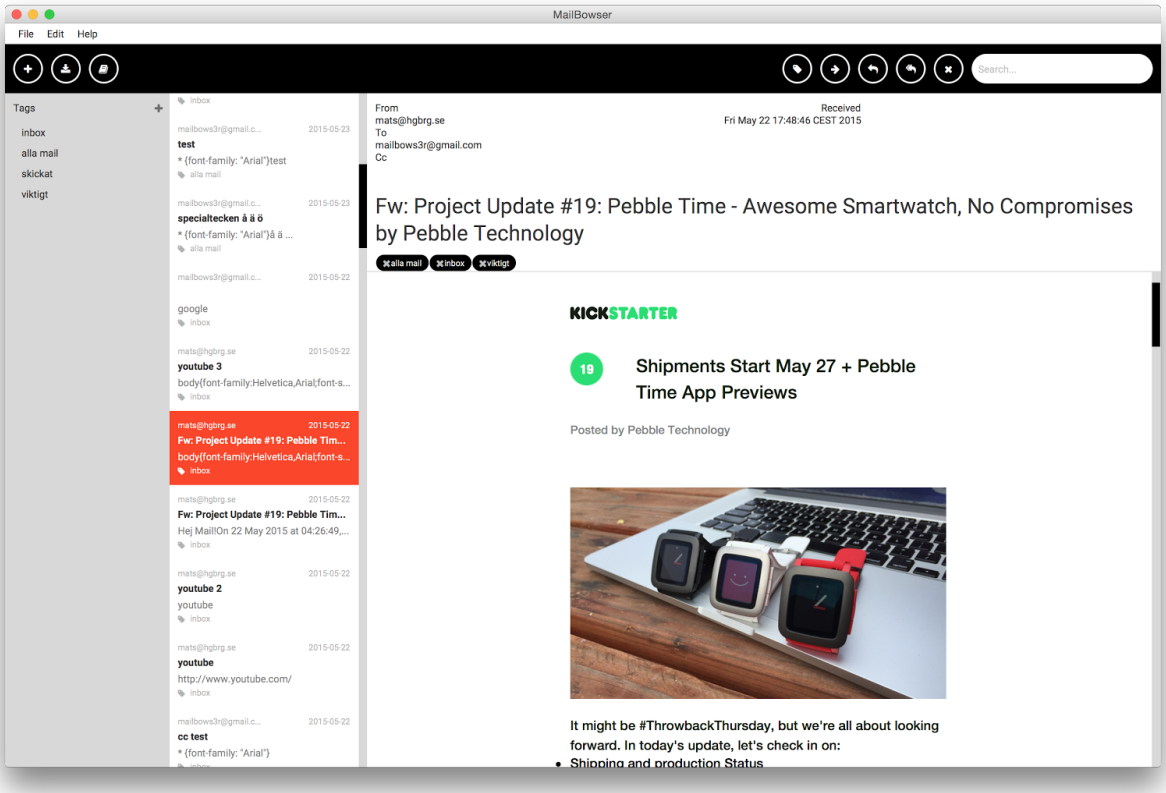


Figure 3: The MailBowser GUI

2.1.3 Models and handlers

The application makes use of two types of classes to handle data - models and handlers. Models represent real world objects, and contain primitive data and/or other model objects. All the back end application logic is implemented in these models.

Handlers on the other hand are classes that only store other model objects and their relationship to each other. For example there is an account handler for storing accounts, and a tag handler for storing the relation between tags and emails. If you want to add an account to the application, you add it to the account handler, and if you want to tag an email with a specific tag, you do it via the tag handler. The application also has a singleton class called MainHandler that is used to keep track of the different handlers. This is to make the handlers

easily accessible throughout the application. One alternative to this would be to make each individual handler into a singleton. This would make the application less modular, since singleton implementations are hard to swap out. They are also hard to test, so this approach was discarded in favour of using a MainHandler.

All the public methods found in models and handlers are defined in interfaces, and these interfaces are used exclusively throughout the program. The code never relies on specific implementations, only on the abstractions that the interfaces provide. This follows the *Dependency Inversion Principle* (Martin, 2003), and makes it easy to replace any concrete implementation of a model with a new implementation.

All models and handlers overrides the equals and hashCode method as per praxis.

2.1.4 JavaMail API

The JavaMail API is used to handle all communication with physical mail servers. The application also uses the API to validate email address.

2.1.5 Account module

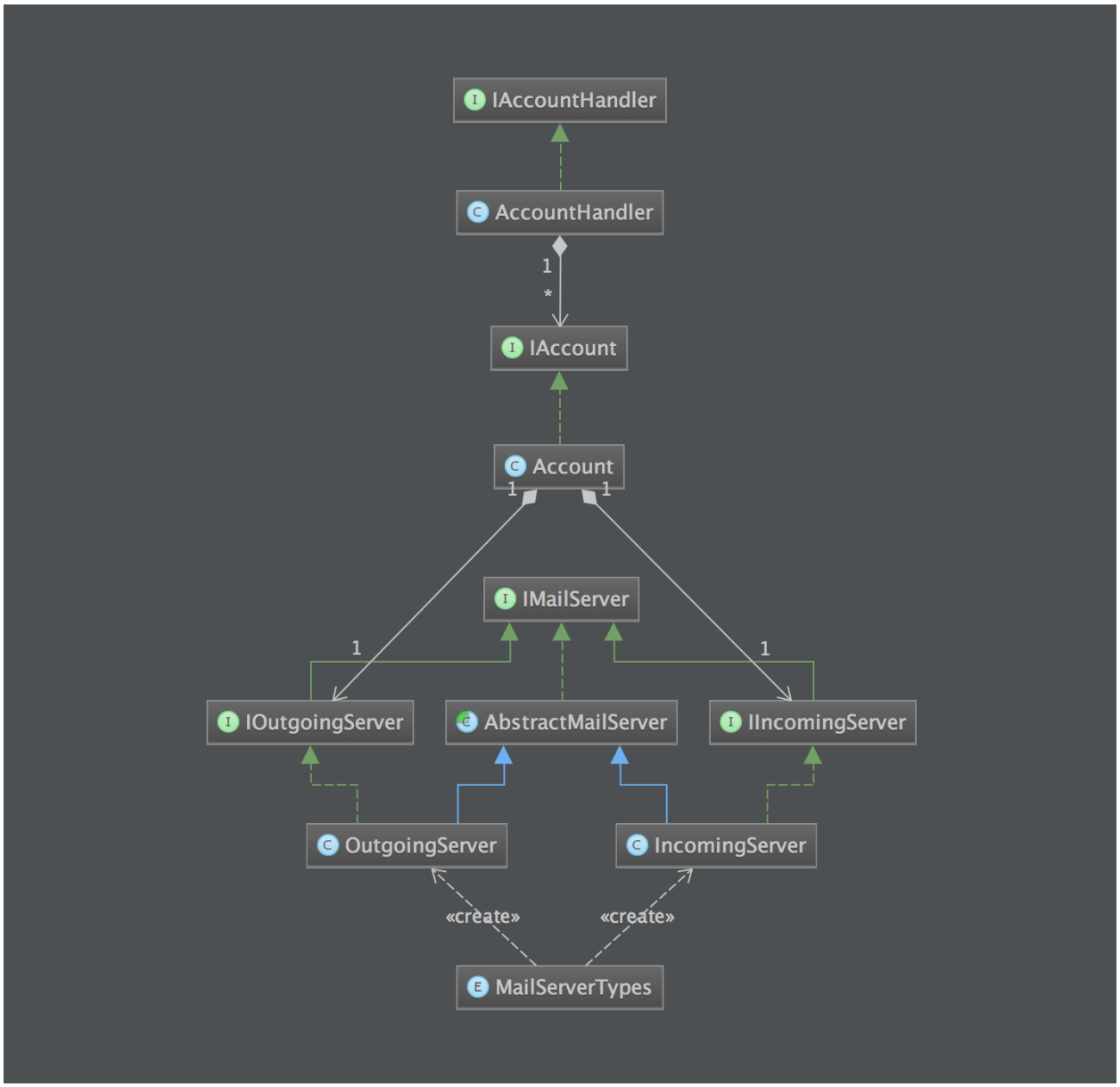


Figure 4: The account module

The account module holds all information and logic needed to send and fetch emails. The main class here is Account, as an object of this class represents a physical email account. An account has two MailServer objects - one for sending emails and one for fetching emails. These classes - OutgoingServer and IncomingServer - both extend the abstract class MailServer since they share a lot of characteristics. All the communication with physical mail servers is done in these two classes, but the Account class acts as an interface between them and the rest of the program. This holds external dependencies to these classes to a minimum,

which in turn makes it easy to replace the specific implementations of sending and fetching if needed.

AccountHandler is a class for keeping track of the accounts that have been added to the application. This class can return a list of all the added accounts, but it also has methods for manipulating all the added accounts at once. For example it has a method for initiating fetching in all added accounts. This helps at following the design principle *Expert Pattern* (Skrien, 2008), since AccountHandler is holding the data and therefore is most suitable for controlling it.

MailServerType is a factory for creating pre-configured mail server objects. This makes it easy to implement different standard account types while still retaining the possibility to create custom accounts without a specific type. This factory is implemented using the modern enum approach as described by Mishra in the article *Factory Design Pattern - An Effective Approach (*2012), as it is both faster and more straight forward to use compared to the standard way of implementing factories.

The password stored in the Account class is never stored in plain text. The setPassword method takes a string, encrypts it and stores in an array of bytes. The getPassword method does the opposite - it takes the stored array of bytes and decrypts it with the same key that was used in the encryption. The result is the original string that was sent to the setPassword method before. This is according to the *Information Hiding Principle ,* which says that a modules internal implementation should be hidden from other modules, and that external access should be provided through a well defined interface. Other modules simply see the password as a string that they can set and get, and don't have any knowledge about how the password is actually stored.

2.1.6 Sending and Fetching Email

Sending and fetching emails involves communicating with physical mail servers over the Internet, and these requests can be very slow. The sending and fetching of emails is therefore done asynchronously in new threads to prevent the GUI from freezing during these requests.

This is done with internal classes in OutgoingServer and IncomingServer called *Sender* and *Fetcher* that implements the *Runnable* interface. When sending an email, a new Sender object is created and started in a new thread. All requests to the physical mail server is done by this object.

Since the sending and fetching is done asynchronously, the code execution will move on after the method call. Therefore it is not possible to simply return a value when the process is finished. To solve this problem, a callback is sent in as a parameter to the method. This callback holds information about what action to perform on success or failure. This method is described by Skrien in *Object-Oriented Programming Using Java* (2008) as the *Command Pattern*.

The method used when fetching emails asynchronously is the same as when sending an email, but the internals of the fetch is a bit more complicated. Emails are stored in folders on the server, and each folder can contain both emails and other folders. To fetch all emails for an account, the root folder for that account is fetched first. All subfolders are then recursively worked through until a folder that doesn't contain other folders are found. On each iteration the folder is checked for emails, and in the end a list of all the fetched emails from all folders is returned.

2.1.7 Contact book

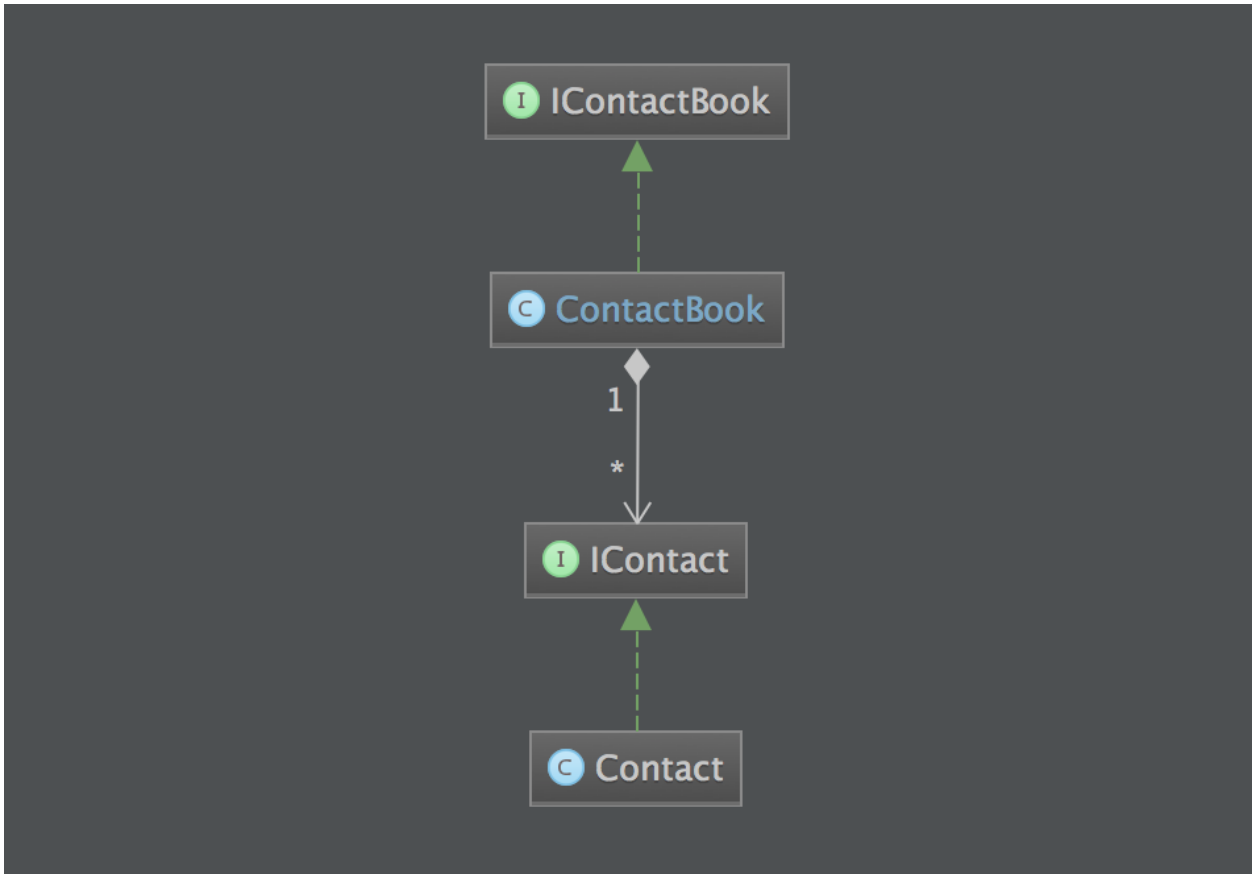


Figure 5: The contact book module

The contact book contains two models - ContactBook and Contact. The ContactBook class is essentially a handler, as it is only responsible for keeping track of contacts. Its behaviour closely matches a real world contact book, and therefore it is called ContactBook instead of ContactHandler. A ContactBook object can contain many contacts, and each contact has a name and a list of addresses.

2.1.8 Email

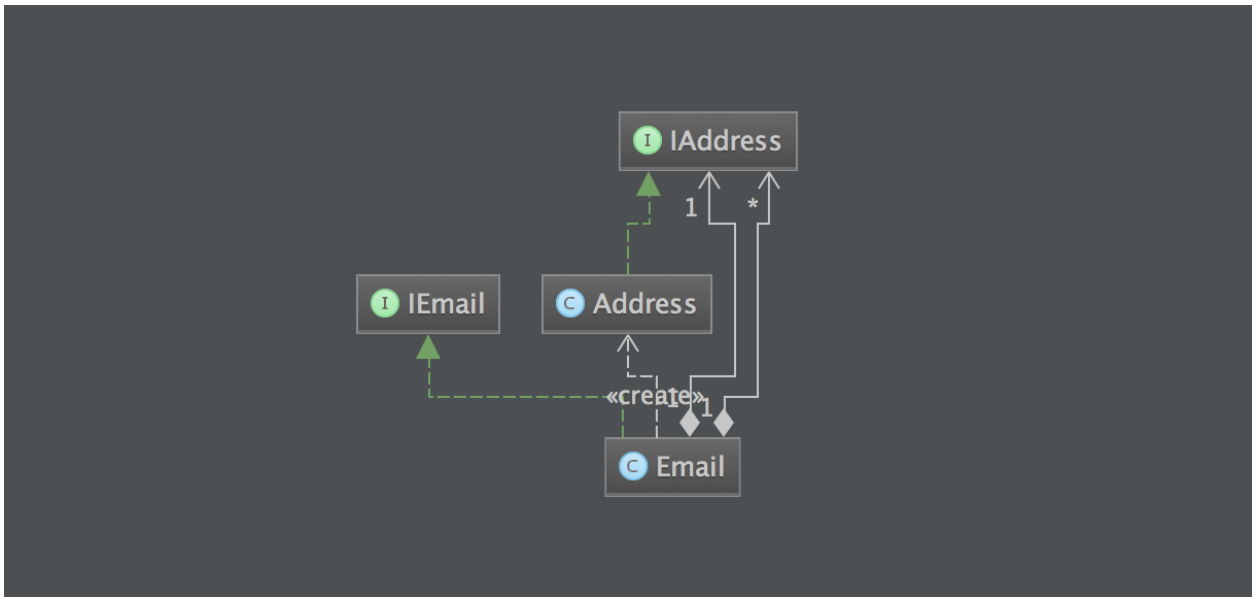


Figure 6: The email module

The email module is very simple. It only contains two classes that have no particular functionality. Both the Address class and the Email class are only used for storing information. This package is tightly coupled with the JavaMail API. Both the Email and the Address class have constructors for creating instances from their JavaMail counterparts, and methods that returns JavaMail objects from existing Email and Address instances. This is all according to the Information Expert Principle described in section 2.1.5.

Emails have a lot of fields that are optional. Instead of providing constructors for all the different field combinations, a *builder* is used as described by Bloch in *Effective Java* (2008). This builder enables incremental building of Email objects, while still ensuring that no incomplete Email objects are instantiated.

2.1.11 Tag

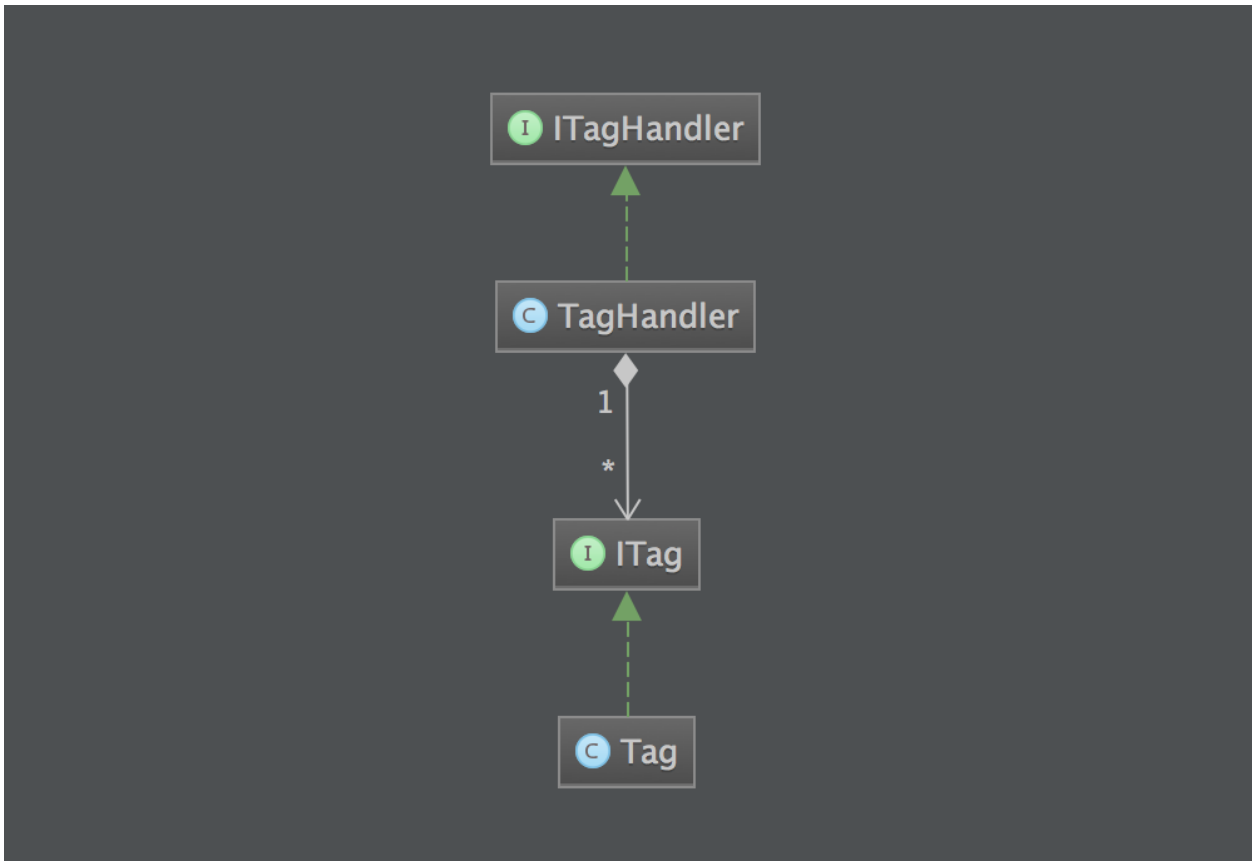


Figure 7: The tag package

The tag package consists of two classes, TagHandler and Tag. Tags are used in the application to sort emails. An email can have many tags, and many emails can have the same tag.

Tag is a simple class for storing information about a tag. TagHandler is responsible for keeping track of the relationship between emails and tags. The tag handler acts as a layer between the Tag and Email classes, and prevents circular dependencies. An alternative to this would be to keep a list of tags in each email, a list of emails in each tag. This would increase the risk of synchronization issues since the same data would be stored in two separate places, and also cause the Email and Tag classes to be tightly coupled.

TagHandler stores the relationship between emails and tags in two Map objects. One map with tags as keys and sets of emails as values, and one with emails as keys and sets of tags as values. The two maps are synced whenever a change occurs, to ensure that they don’t differ.

Another option that would have solved circular dependencies would have been to let a tag have a set of emails and use a tag handler that only keeps a map with emails as keys and sets of tags as values. The reason to choose the now implemented option is to have an as simple Tag class as possible.

2.1.12 Utils

The utils module contains convenience classes and methods that are completely independent from the rest of the application. This module is mainly for minimizing code duplication in accordance to the *D.R.Y. principle* (Skrien, 2008).

2.1.12.1 Search

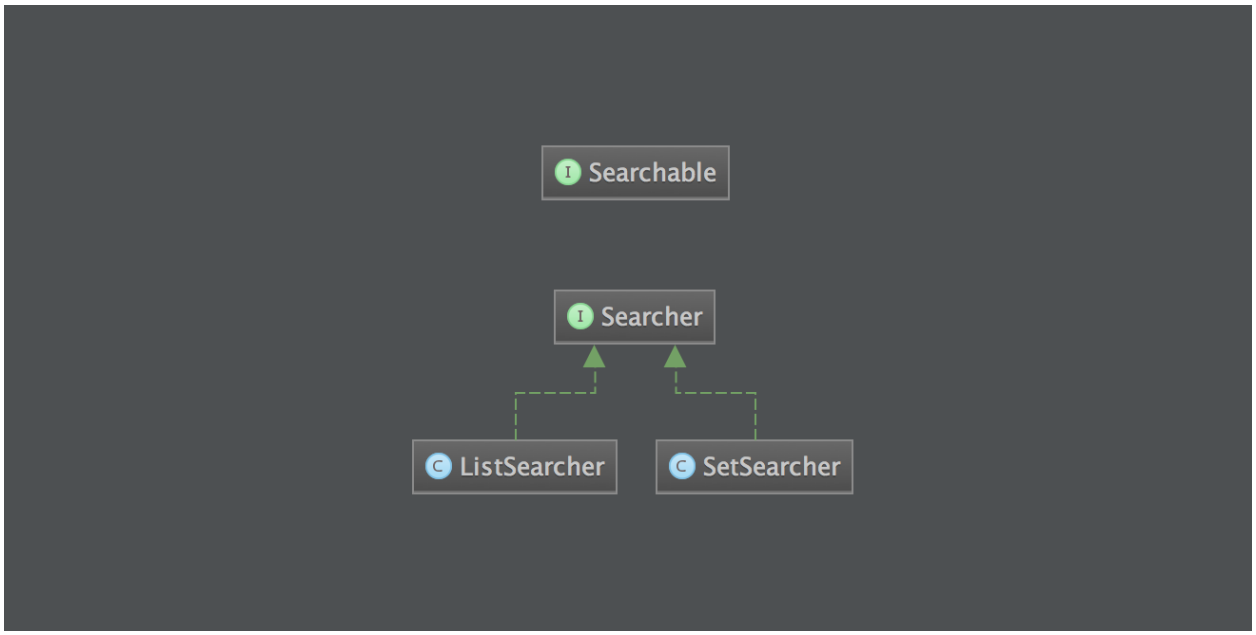


Figure 8: The search utility package

The search utility is for searching in collections. It contains two interfaces - Searchable and Searcher. The Searchable interface is a functional interface that only has one method: boolean matches(String query). This method determines if an object matches a given query.

A searcher takes a collection of Searchables and a query string, and returns a new collection containing all the items that matches the query. It is up to each individual item in the collection to determine if it matches the query or not. Currently there are two implementations of Searcher, but more can be added by creating new classes that implements the Searchable interface. This follows the *Open-Closed Principle* (Skrien, 2008) in that existing classes don't have to be modified in order to add new functionality.

2.1.12.2 I/O

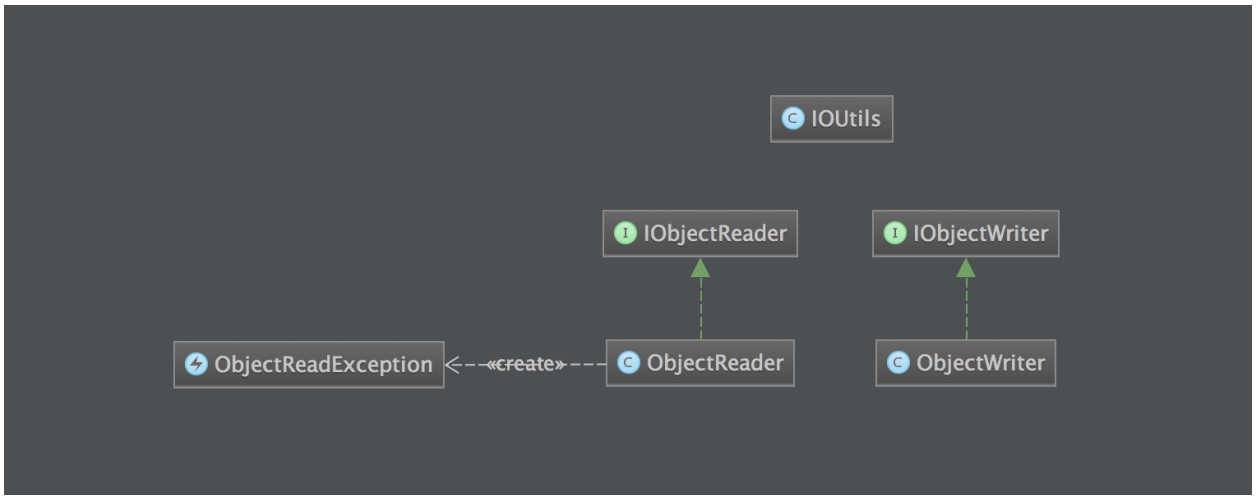


Figure 9: The I/O utility package

The I/O utility makes it easy to handle Seralizable objects. The two main classes - ObjectReader and ObjectWriter - are generic classes that reads respectively writes objects to and from disk.

2.1.13 Periodic fetching

To achieve periodic fetching of new emails a timer task called *FetchTask* is used. A timer with a set interval is started in Main with this task, and it continues to run until the application is closed.

2.1.14 Event

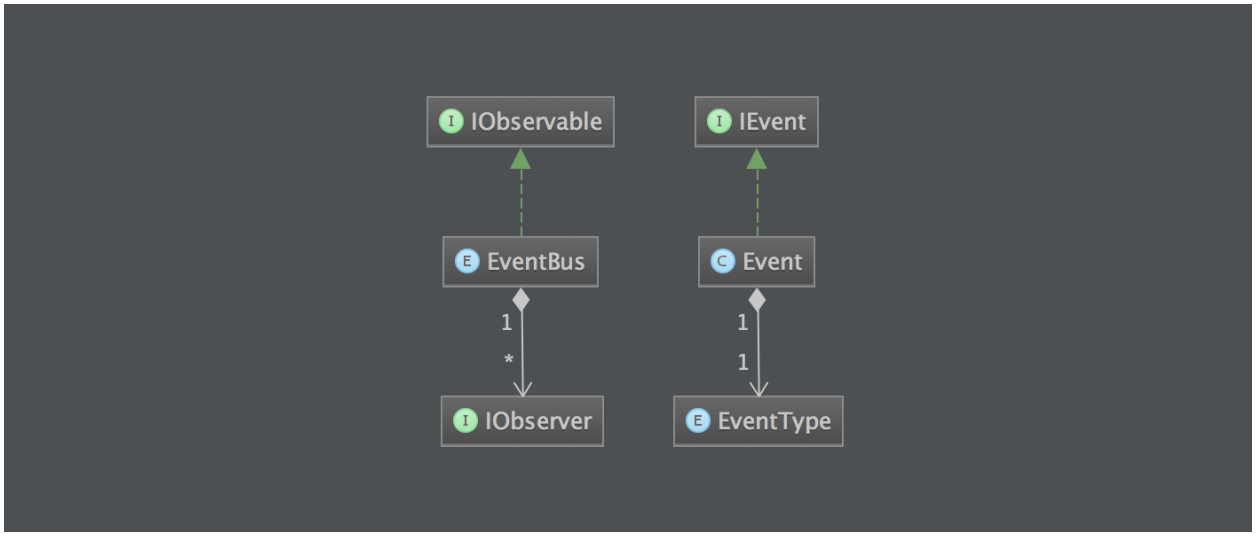


Figure 10: The event package

Events consists of a type and a value. All the available event types are specified in the EventType enum. This makes it easy to get an overview of all the different event types used in the application, in contrast to using a simple string to identify event types.

2.2 Software decomposition

2.2.1 General

The application is decomposed into the following modules (see figure 11):

- account - Contains classes need to connect to real servers and handles all accounts.
- contact - Contains classes to model and handle contacts.
- email - Contains classes to model emails and email addresses.
- event - Handles the communication between classes, event module is based of the concept of event bus.
- presenters - Contains the presenter logic. Presenter parts for MVP.
- tag - Contains classes to model tag and handle tags.
- utils - Contains classes that provide convenience methods
    - utils.io - Contains classes to read and write objects to the disk
    - utils.search - Contains classes to search collections for matching objects

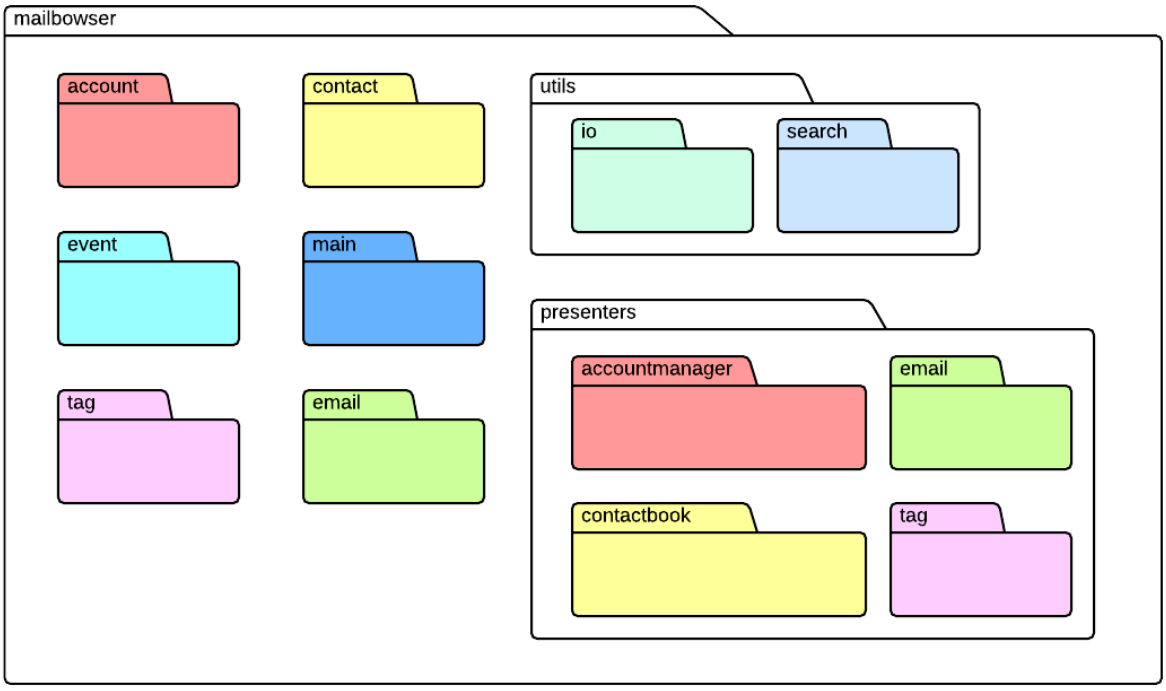


Figure 11: Package structure

2.2.2 Decomposition into subsystems

The account module is a not unified subsystem that handles all communication with the servers.
The i/o package is also a not unified subsystem but only classes handling the writing and reading from disk.

2.2.3 Layering

The application is highly modular which implies that you can remove the upper packages and still have a working application. For example if you remove the entire contact module the rest of the application will still work with an exception for the presenter and main modules. This is because none of the other modules are dependent on the contact module.

2.2.4 Dependency analysis

There are no circular dependencies (see figure 12).

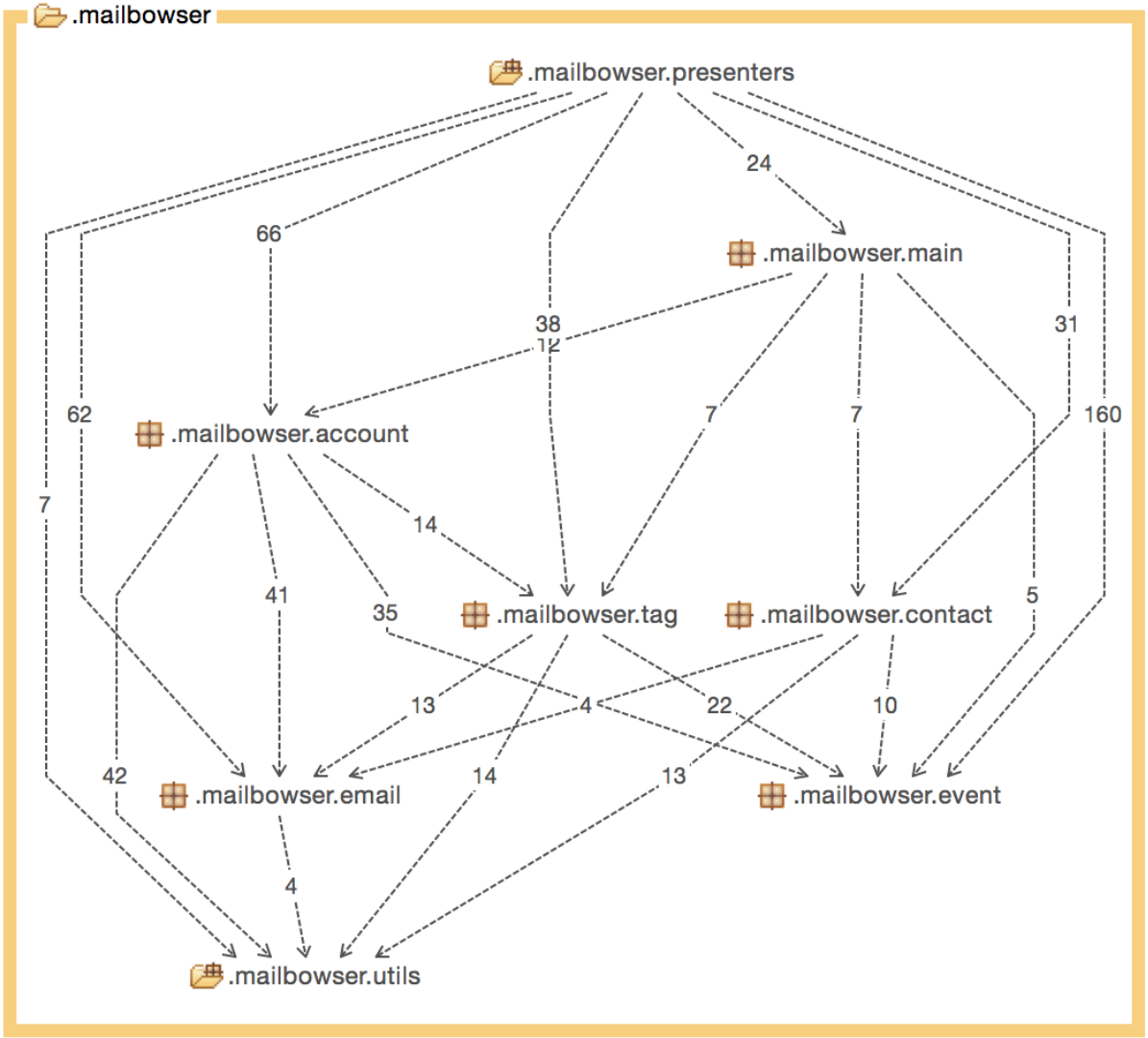


Figure 12: Layering and dependency analysis

2.3 Concurrency issues

The application makes heavy use of concurrency, but it doesn't result in any issues. JavaFX takes care of the GUI in its own thread, and every network request is done asynchronously in separate threads (as described in 2.1.6). Modifier methods in classes that can be modified by multiple threads simultaneously has the *synchronized* keyword to prevent concurrent modification errors. Before accessing the GUI the current thread is synchronized with the JavaFX thread.

2.4 Persistent data management

The application makes use of Java's Serializable interface for persistent data storage. The objects that are stored to disk are contacts, accounts, emails and tags, as well as their relationship to each other. The files that are saved are:

- Accounts.ser - for accounts and their fetched emails.
- Contacts.ser - for all contacts in the contact book.
- Tags.ser - for tags and their relationship to emails.

The writing and reading from disk is done by the handlers.

2.5 Access control and security

- Password encryption (as described in 2.1.5).
- Encryption regarding sending and receiving emails. The application uses mail protocols that support encrypted connections, but these connection types are not implemented in the application yet.

2.6 Boundary conditions

NA. The application will be launched as any standard desktop application and closed as so.

3 References

Bloch, Joshua. (2008) *Effective Java*, 2nd edition, Addison-Wesley

Chandel, Sumit. (2009) *Testing,* Google Development Relations
http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html

Gruber, John. (2014) *Markdown*
http://daringfireball.net/projects/markdown/

Martin, Robert C. (2002) *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall

Mishra, Debadatta. (2012) *Factory Design Pattern - An Effective Approach*, DZone
http://java.dzone.com/articles/factory-design-pattern

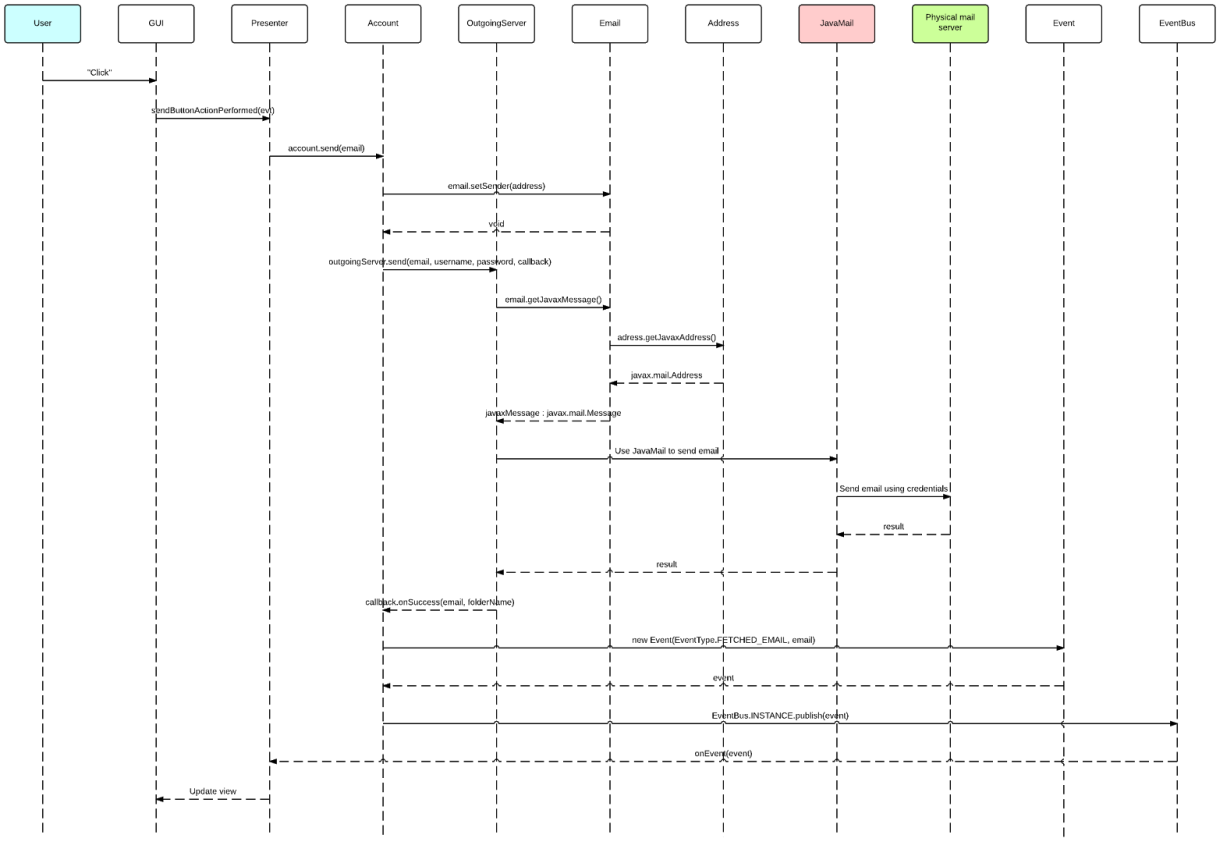
Oracle. (2015) *JavaMail API*, Oracle Technology Network
https://javamail.java.net/nonav/docs/api/

Skrien, Dale. (2008) *Object-Oriented Design Using Java*, McGraw-Hill
http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter

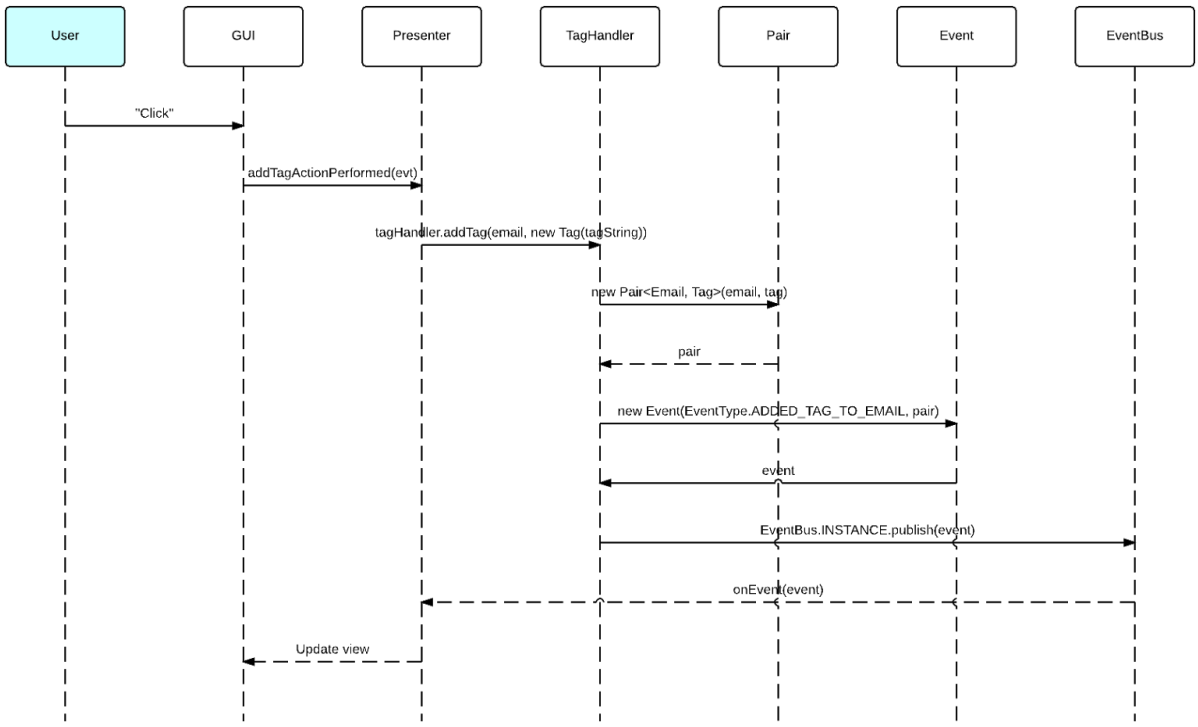
Appendix

Sequence diagrams:

Send email



Tag email


User
GUI
Presenter
TagHandler
Pair
Event
EventBus
"Click"
addTagActionPerformed(evt)
tagHandler.addTag(email, new Tag(tagString))
new Pair<Email, Tag>(email, tag)
pair
new Event(EventType.ADDED_TAG_TO_EMAIL, pair)
event
EventBus.INSTANCE.publish(event)
onEvent(event)
Update view