SeaMail The best command-line email client under the sea.

Introduction

We set on to create a command-line email client for the 21st century only to be humbled by the scope of such an endeavour. The specification offered by Rob provided us a map of expectations which we tried to span through our initial work. Despite the reduced scope, we felt anemic on time given the level of user interaction we needed to support. It was thrilling then when we finally achieved full fidelity (relatively speaking) product. What follows documents our journey working on SeaMail v1.

Overview

We structured SeaMail in the spirit of Model View Controller (MVC), with a distinct backend and frontend. The backend provided the Account subjects, which were observed by the observer Views implemented in the frontend, implementing the Observer pattern. By using the Observer pattern, our graphical Views are able to respond to changes in the Subject Account immediately (reactive in a sort of way). Controllers hooked on the user interface allowed for mutations on all backend constructs which in turn would notify the Views of the corresponding event. To provide further insights on the constructs we cooked up for this project, here's a high-level overview:

Backend of Backend

- *EmailProvider*, *LocalEmailProvider* (representing a generalized email server such as GMail, where *LocalEmailProvider* is backed by local storage)
- *Serializable* (for persistence)
- *LocalState* (for managing program state)
- Miscellaneous stuff: *Errors* (for coordinating error handling), *AccountEvents*, *Subject*, *Observer* (for implementing the observer pattern)

Backend

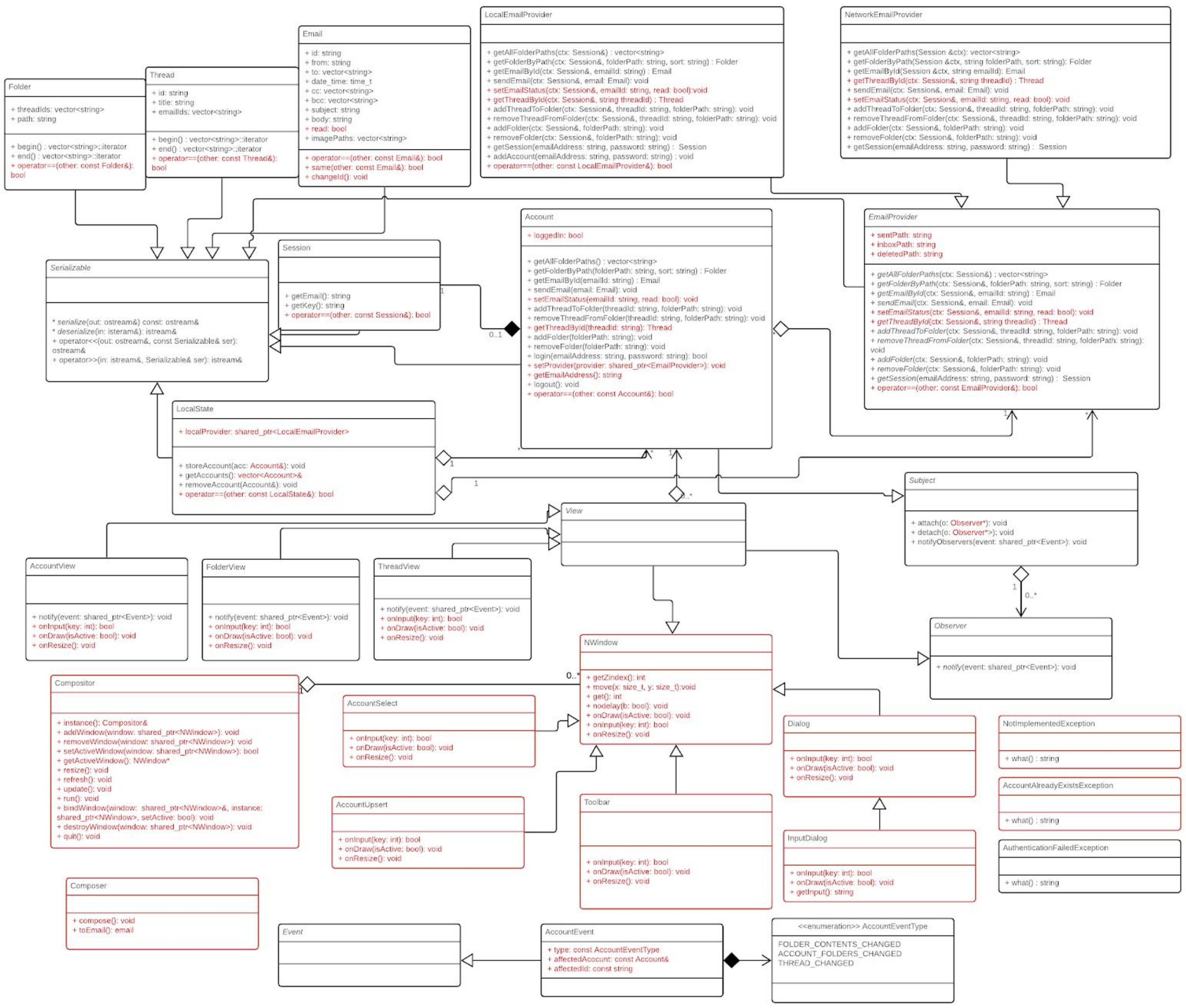
- *Email*, *Thread*, *Folder* (in increasing degrees of aggregation)
- *Account* (derives from *Subject*), *Session* (logged in *Accounts* own a *Session*)

Frontend

- *AccountView* (display folders in *Account*), *FolderView* (display threads in *Folder*), *ThreadView* (display emails in *Thread*)
- *Compositor* (for managing all the *View* windows, overlapping, window switching)
- Miscellaneous stuff (a lot): *Composer* (for using nano to compose emails), *Toolbar*, *Dialog* (for account selection/creation/login)

The following sections should provide an account of the implementation details for the most important & interesting pieces.

Updated UML


Folder
+ threadIds: vector<string>
+ path: string
+ begin() : vector<string>::iterator
+ end() : vector<string>::iterator
+ operator==(other: const Folder&): bool
Thread
+ id: string
+ title: string
+ emailIds: vector<string>
+ begin() : vector<string>::iterator
+ end() : vector<string>::iterator
+ operator==(other: const Thread&): bool
Email
+ id: string
+ from: string
+ to: vector<string>
+ date_time: time_t
+ cc: vector<string>
+ bcc: vector<string>
+ subject: string
+ body: string
+ read: bool
+ imagePaths: vector<string>
+ operator==(other: const Email&): bool
+ same(other: const Email&): bool
+ changeId(): void
LocalEmailProvider
+ getAllFolderPaths(ctx: Session&) : vector<string>
+ getFolderByPath(ctx: Session&, folderPath: string, sort: string) : Folder
+ getEmailById(ctx: Session&, emailId: string) : Email
+ sendEmail(ctx: Session&, email: Email): void
+ setEmailStatus(ctx: Session&, emailId: string, read: bool):void
+ getThreadById(ctx: Session&, string threadId) : Thread
+ addThreadToFolder(ctx: Session&, threadId: string, folderPath: string): void
+ removeThreadFromFolder(ctx: Session&, threadId: string, folderPath: string): void
+ addFolder(ctx: Session&, folderPath: string): void
+ removeFolder(ctx: Session&, folderPath: string): void
+ getSession(emailAddress: string, password: string) : Session
+ addAccount(emailAddress: string, password: string) : void
+ operator==(other: const LocalEmailProvider&): bool
NetworkEmailProvider
+ getAllFolderPaths(Session &ctx): vector<string>
+ getFolderByPath(Session &ctx, string folderPath, sort: string): Folder
+ getEmailById(Session &ctx, string emailId): Email
+ getThreadById(ctx: Session&, string threadId) : Thread
+ sendEmail(ctx: Session&, email: Email): void
+ setEmailStatus(ctx: Session&, emailId: string, read: bool): void
+ addThreadToFolder(ctx: Session&, threadId: string, folderPath: string): void
+ removeThreadFromFolder(ctx: Session&, threadId: string, folderPath: string): void
+ addFolder(ctx: Session&, folderPath: string): void
+ removeFolder(ctx: Session&, folderPath: string): void
+ getSession(emailAddress: string, password: string) : Session
Serializable
* serialize(out: ostream&) const: ostream&
* deserialize(in: isteram&): istream&
+ operator<<(out: ostream&, const Serializable& ser): ostream&
+ operator>>(in: istream&, Serializable& ser): istream&
Session
+ getEmail(): string
+ getKey(): string
+ operator==(other: const Session&): bool
Account
+ loggedIn: bool
+ getAllFolderPaths() : vector<string>
+ getFolderByPath(folderPath: string, sort: string) : Folder
+ getEmailById(emailId: string) : Email
+ sendEmail(email: Email): void
+ setEmailStatus(emailId: string, read: bool): void
+ addThreadToFolder(threadId: string, folderPath: string): void
+ removeThreadFromFolder(threadId: string, folderPath: string): void
+ getThreadById(threadId: string): Thread
+ addFolder(folderPath: string): void
+ removeFolder(folderPath: string): void
+ login(emailAddress: string, password: string): bool
+ setProvider(provider: shared_ptr<EmailProvider>): void
+ getEmailAddress(): string
+ logout(): void
+ operator==(other: const Account&): bool
EmailProvider
+ sentPath: string
+ inboxPath: string
+ deletedPath: string
+ getAllFolderPaths(ctx: Session&) : vector<string>
+ getFolderByPath(ctx: Session&, folderPath: string, sort: string) : Folder
+ getEmailById(ctx: Session&, emailId: string) : Email
+ sendEmail(ctx: Session&, email: Email): void
+ setEmailStatus(ctx: Session&, emailId: string, read: bool): void
+ getThreadById(ctx: Session&, string threadId) : Thread
+ addThreadToFolder(ctx: Session&, threadId: string, folderPath: string): void
+ removeThreadFromFolder(ctx: Session&, threadId: string, folderPath: string): void
+ addFolder(ctx: Session&, folderPath: string): void
+ removeFolder(ctx: Session&, folderPath: string): void
+ getSession(emailAddress: string, password: string) : Session
+ operator==(other: const EmailProvider&): bool
LocalState
+ localProvider: shared_ptr<LocalEmailProvider>
+ storeAccount(acc: Account&): void
+ getAccounts(): vector<Account>&
+ removeAccount(Account&): void
+ operator==(other: const LocalState&): bool
Subject
+ attach(o: Observer*): void
+ detach(o: Observer*>): void
+ notifyObservers(event: shared_ptr<Event>): void
View
Observer
+ notify(event: shared_ptr<Event>): void
AccountView
+ notify(event: shared_ptr<Event>): void
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
FolderView
+ notify(event: shared_ptr<Event>): void
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
ThreadView
+ notify(event: shared_ptr<Event>): void
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
NWindow
+ getZindex(): int
+ move(x: size_t, y: size_t):void
+ get(): int
+ nodelay(b: bool): void
+ onDraw(isActive: bool): void
+ onInput(key: int): bool
+ onResize(): void
Compositor
+ instance(): Compositor&
+ addWindow(window: shared_ptr<NWindow>): void
+ removeWindow(window: shared_ptr<NWindow>): void
+ setActiveWindow(window: shared_ptr<NWindow>): bool
+ getActiveWindow(): NWindow*
+ resize(): void
+ refresh(): void
+ update(): void
+ run(): void
+ bindWindow(window: shared_ptr<NWindow>&, instance: shared_ptr<NWindow>, setActive: bool): void
+ destroyWindow(window: shared_ptr<NWindow>): void
+ quit(): void
AccountSelect
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
AccountUpsert
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
Toolbar
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
Dialog
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ onResize(): void
InputDialog
+ onInput(key: int): bool
+ onDraw(isActive: bool): void
+ getInput(): string
NotImplementedException
+ what() : string
AccountAlreadyExistsException
+ what() : string
AuthenticationFailedException
+ what() : string
Composer
+ compose(): void
+ toEmail(): email
Event
AccountEvent
+ type: const AccountEventType
+ affectedAcocunt: const Account&
+ affectedId: const string
<<enumeration>> AccountEventType
FOLDER_CONTENTS_CHANGED
ACCOUNT_FOLDERS_CHANGED
THREAD_CHANGED
1
0..1
0..*

As can be seen in the updated UML diagram above with changes highlighted in red, many unforeseen changes needed to be made. On the backend, smaller changes like the addition of equality operators and some required functions on *Account* were missing. Since we didn't know what integrating Ncurses on our frontend would entail, all of the UI-related objects were added and inherit from the *NWindow* class. Also, the *EmailView* was replaced with *Composer*, which pops up an email viewer/editor using less/nano instead of a custom view.

Design

The scope of our project was overwhelming at first. When we saw our UML diagram in comparison to folks doing Tetris, we had our moments of doubt. Our naive optimism led us to soldier on, telling the haters in our heads to hate. Retrospectively, we're glad for that voice because we all came out learning new things and feeling better for having brought SeaMail to life. Here's some of the more remarkable challenges we faced:

- **Building an email client in 2 weeks**: The solution was to sit our asses down and code every day. No design pattern could've saved us for the lack of the prior.

- **User Interface**: Building a TUI that would feel natural for users to interact with was quite a challenge. While the ncurses library helped in the mechanical portions of drawing something on the screen, we still needed to design a way to create and manage all our views.

  The *NWindow* ADT represents generalized version of a tiling Window in the traditional sense, backed by an ncurses window. *NWindow* contains virtual member functions related to drawing, input handling, and resizing, which are overridden by derived classes to exhibit configurable behaviour. These *NWindows* are managed by a *Compositor*, which handles draw order and input arbitration so that *NWindows* are able to tile and focus. *Compositor* implements the singleton design pattern, as it requires exclusive control over the active terminal window such that multiple instances are not logical.

- **Persistence**: It is not exactly trivial to have a non-trivial program state last beyond execution. The serialization and deserialization need to be invertible functions. But strategy pattern was deployed to the rescue (aka virtual dispatch). The abstract base class *Serializable* overloads input and output operators, which in turn call pure virtual protected functions deserialize and serialize. *Serializable* utilizes the non-virtual interface design idiom by separating the private serialization methods from its usage through the input/output operators, allowing for changes to its implementation without changes to its interface. Each of the concrete classes implement these serialization methods. In the case of composition or specialization, serialization is done in chains such that a top-level *Serializable* object store/restores all of its members along with itself. I don't think the power of virtual dispatch could become any more explicit when you are able to revive your entire program state based using just one function that can recursively called (but differently because different object calling it). Also, huge s/o for nlohmann's json library. This would have been a lot more painful without it.

- **Ownership**: This follows directly from persistence because you need to be very explicit on what owns what else you'll have things randomly disappear on restarts. Singleton pattern came in handy again; this time with *LocalState*. *LocalState* owns all *Accounts* and all the *EmailProviders* which in turn own everything of concern. So, just serialize *LocalState* and the persistence logic talked in the previous point will take care of deserializing everything else (sorta) recursively. *LocalState* also serves another important function of being the single source of truth for the program state for Views, the main loop, controllers etc. If your account doesn't exist on *LocalState* even though some side class decided to take it upon itself to tell the user the account was added, then it doesn't exist, end of story, too bad for that liar of a class - bad class, no cookie for you.

- **Contracts**: We talked a bit before about the frontend, the backend and the backend of backend in our due date 1 document. This separation arose from the big vision idea that we saw each component fulfill. It's akin to how a RESTful API provides a contract with the frontend. Contracts are pretty much adult, coding version of pinky promises. Our backend to backend provided a contract to the backend that says don't worry about the email provider implementation; the expected actions of the email provider will be done. Our backend provides a contract that to the frontend that ensures it can rely on *Accounts* for all its data. This idea of interface specification was incredibly helpful in writing a clean codebase.

  The required behaviour for seamail's operation is defined by the abstract interface of an EmailProvider (i.e. at a bare minimum, an email provider must be able to provide certain functionality such as sending emails, listing folders etc.) An Account utilizes an EmailProvider based upon the contract defined between them, such that there is no dependency on the actual EmailProvider implementation. As a result, this design is extensible, as any EmailProvider (e.g. GMail, Outlook, local storage etc.) can be dropped in as long as the appropriate specialization of EmailProvider is implemented.

- **Facade Pattern**: *Accounts* are pretty much empty constructs. If you check in the implementation of *Account*, all the calls are redirected to the corresponding call in the *EmailProvider*. While this might seem redundant, there-in lies the beauty of facade pattern. Theoretically, we could have had a much more awful interface provided by *EmailProvider* or maybe some missing feature. In case of such an *EmailProvider*, we would have had to completely rework our *Views* in absence of *Account*. Now, *Account* pinky promises the frontend so it must continue to support the interface of `sendEmail`, `getFolderByPath`etc. despite what *EmailProvider* it might come across.

  The relationship between Account and EmailProvider is an additional layer of abstraction which makes the client-Account interface more resilient to change. Account supports additional behaviour (such as authentication, event handling, and observation) which cannot be generalized across all EmailProviders but is required for seamail's functionality. Account provides this behaviour as a shim (akin to NVI), passing through and composing EmailProvider operations to provide the required interface to graphical client code. An example of this is the `sendEmail` endpoint, which must execute the operation in addition to notify observing members of the change in state.

- **Observer Pattern**: The *Observer* pattern is great and is implemented in our project it in its full glory (with differential Event handling etc.). *Account* (*Subject*) is observed by multiple *Views* (*Observers*). There occurs an *AccountEvent* (*Event*) when we access *Account* and change the program state. *Views* choose how they respond to the event given its type in *AccountEventType*.

This design allows graphical content to respond immediately to changes in state, in a performant way. The graphical content will not redraw unless a change to state occurs. Additionally, this design allows new graphical constructs to be integrated easily, as any new view can observe its relevant Account and respond to any changes in Account state.

- **Composition**: Again, this is pretty fundamental so we can probably see this in a lot of places in our project. One notable mention is relationship of *Session* and *Account*. *Account* should have been specialized to *LoggedInAccount* upon login; however, that seems a bit silly. It's much more natural to compose an *Account* with either an empty *Session* or logged in one containing all the deets. The utility of composition over inheritance is undeniable in this case.

Resilience to Change

We technically talked about a lot of this in the previous section. Oh well, let's keep talking. There's lots of fun stuff left to discuss.

- **Test Suite**: GCC error messages are awful. Any noob wanting to build something scalable in C++ will soon realize that. The fact of the matter is there's a lot of moving pieces in these projects and it's easy to cause a regression with a simple change without even realizing it. Our decently comprehensive test suite provides us resilience to change without regression.

  The design is compartmentalized (highly cohesive, low coupling), allowing for components to be tested and verified separately without functioning implementations for the entire program.

- **High Cohesion, Low Coupling**: This concept can be seen applied to our project on many levels. Let's begin with the biggest scope and progress to the module level. Our project has a clear separation of the responsibilities of frontend, backend, and backend of backend. Technically you could switch out one for another as long as the contract (interface specification) is met, thereby implying low coupling. Since each of these parts satisfy a very clearly defined shared purpose, we have high cohesion.

  Next, looking at the frontend, we have *Compositor* which manages *NWindows*. Each *View* implements its own functions related to *NWindows* and *Compositor* is able to manage all of them through a few functions in their public interface thereby implying low coupling. We have high cohesion because we have very defined purpose for the class of *NWindows*.

  The graphical portion of seamail is highly cohesive, in the sense that the code is not specific to seamail and can be reused in some other graphical program. NWindow represents the abstract concept of a tiling window, which is implemented and derived by seamail specific views which draw content depending on account state.

- ***EmailProvider***: This also ties in with High Cohesion, Low Coupling. The specification required us to create a local version of SeaMail (in retrospect, phew). However, we knew early on that we wanted to make this into an actual usable email client which means supporting addition of accounts from Gmail, Outlook etc. For the purpose of CS247, we implemented *LocalEmailProvider*; however, it's very clear that we can create a *GmailEmailProvider* or *IMAPEmailProvider* to support more email providers - all of them just need to derive them *EmailProvider* and fulfill that functionality. Clearly, this is pretty extensible as intended.

  The abstract interface between EmailProvider and Account allows any implementation of EmailProvider to function with seamail as a complete unit, as long as the basic requirements of an EmailProvider are met. As a result, the client interface of an Account (as used by graphical views, and the controller) are not coupled to the implementation of a specific email provider.

- ***AccountEvent***: We can imagine how big one could make this project. In that case, sophisticated event handling will be the key to performance. If there's just one event with `notifyObservers` then all views would have to redraw - imagine redrawing the entire canvas for every action that the user performs. Clearly, this is not ideal. Having setup the foundation of *AccountEvent*, one can avoid such a situation down the line by making a more sophisticated event handling architecture on the frontend.

  At any time, new views implementing new graphical functionality can be added without affecting the implementation of an Account. A graphical view may attach itself to an Account (as all views are observers to account), and obtain any required information for its functionality when an Account emits a certain event.

In general, the project is pretty extensible. If there's one functionality that annoys you, you probably only need to make a change in one place to amend that behaviour because our classes are well laid out in terms of how they interact and specialized enough to support this.

Meeting Project Requirements

- Users should be able to create a new account. Info about that account should be stored locally somewhere that persists past the execution of the program (i.e. a file)

The frontend facilitates new account creation. The flow will look something like:

User adds new *Account* with credentials -> *Account* constructed -> User clicks the account -> *Session* created attached to *Account* -> *AccountView* is displayed -> User can add new *Folders*, create *Emails* etc.

Backend of Backend will take care of storing account information locally by serializing the client objects (*Serializable*), which includes accounts and deserializing them on restart (described in design).

- Users should be able to logout, taking them back to a main screen where other users can login, or new accounts can be created.

Controller lets *Account* know of logout -> *Session* dropped & *Views* notified -> *Views* dropped.

- Users should be able to login and view their mail, including sorting by predicates of unread and date.

Controller knows of all *Accounts* through *LocalState* -> Controller calls login on selected account -> *Session* created and attached to *Account* on successful authorization by *Account* with *EmailProvider* -> Controller had created the *Views* and attached to the subject *Account* -> *Account* notifies its observer *Views*.

- Users should be able to send email to other users that have been created locally.

Controller tells logged-in *Account* of newly composed email -> *Account* asks its *EmailProvider* to handle it (*LocalEmailProvider* in case of local) -> *LocalEmailProvider* will store the email in a datastore locally -> subject *Account* notifies its observer *Views* of *AccountEvent* -> *Views* update (sent folder has one new email if successful).

Once we log into the receiving account, *Views* observing the *Account* pull its information -> *Account* pulls the information from its *LocalEmailProvider* -> *LocalEmailProvider* pulls the info from its datastore for the logged in account -> *Views* will show the received email for the *Account*.

- Users should be able to view the email in the client (no need for any full markup language plaintext is fine).

The *Email* object has the content of the email. Viewing *Emails* uses the *Composer* to launch ‘less’ on a text file template prepopulated with all the email data.

- Users should be able to receive/send image attachments, which they can choose to display in an external window when viewing an email that includes one.

Images attached in emails are image paths wrapped in special string that the client understands is an image. *EmailProvider* handles downloading/sending the image at the path. Users can open the image in an external window when in *EmailView*, this is done by executing ‘display’ on each downloaded image. In the case of *LocalEmailProvider*, the local file paths are used.

- Should have an inbox folder and sent folder that received emails go to and sent emails go to, respectively.

The *EmailProvider* by default provides both the sent and inbox folders and when emails are sent, they are placed in the appropriate folders on both accounts.

- Users should be able to create folders and move emails into those folders.

From the frontend, a keyboard shortcut exists in *ThreadView* to move the thread into another folder. *Folders* hold an array of *Thread* IDs, so we simply remove from one folder’s list and add to the other’s.

- All e-mail info should persist past the execution of the program. That is the program should be able to be quit, and then the user can start it back up and login and view all their emails.

Backend of Backend handles this through serializing the program state for all *Accounts*, *Providers*, and all associated instances and deserializing it back up on restart.

- Should provide message chains as a single merged entity.

Threads exist for this purpose. EmailProvider will handle construction/rearrangement of threads when emails are added/removed from a folder based on the email subject and date.

Extra Credit Features

- Challenge: Making an email client (albeit only a local version)
- Solution: Just do it.

We don't handle any memory on our own so at least 4/10 on this and whatever you wanna bless us with for completing the email client. We also added themes.

Final Questions

What lessons did this project teach you about developing software in teams?

Developing in teams is great. Working alone on a project of our scope would have been quite a herculean undertaking. We learned that if we set up the right processes for collaboration (git etiquette, formatting, talking etc.), it's smoother sailing from there on. Also, we never worked on this alone as in we had 3-4 hour sessions almost everyday for the past 2 weeks to meetup in person. This was quite a unique experience and taught us the difference between working alone vs working together - working together is much faster and smoother if the chemistry is right. While an individual's productivity might be lower in such a circumstance, especially if they know more about whatever work is being done, the group productivity (sigma of individual productivity) is much higher.

What would you have done differently if you had the chance to start over?

Quadris. Jokes aside, I don't think that's quite true. We still would have done SeaMail. Because the specifications had room for interpretation, retrospectively, made some choices that led to a lot more work than ideal - this was hard to foresee for Rob when making the specifications and hard to foresee for us after reading the requirements. A meeting with Rob to discuss requirements beforehand would have been beneficial. For example, the frontend windows and user interactions were a massive portion of the project. It would have been a lot simpler and nicer to code up a REPL-like email client. It's not even clear if the specifications strictly required it though we distinctly remember seeing the specifications and feeling this is the way we need to implement it.

Conclusion

The best part about this project was that we all pushed our boundaries on how much (mostly good) code we could output in a short period of time on a project. The sessions were super productive. We're happy with what we created and hope the reader shares the feeling.