DEMO LINK: https://youtu.be/et0FfEqWj08

FEATURES:

- **Enhanced user messaging experience:** Can be marked as read, unread. Messaged can also be archived (moved to a separate view) or deleted (completely from the system)
- **GUI:** Implemented the MVP model for the GUI using Java Swing.
- **Schedule exporting:**
    - Using this feature, users can generate a schedule website that can be accessed anywhere.
    - How it works:
        - User presses the print button
        - A JSON of events is generated and sent to the server through a POST request
        - The server processes this http request and saves a copy of the JSON
        - When the user accesses the app URL, this is akin to a GET request, which the server responds to with the calendar webpage
        - This webpage then makes an API call to the server requesting the user's JSON, which is the one that we sent before. So the infrastructure is in place to make this a RESTful-type API
        - The webpage then generates a calendar based on the fetched JSON
- **Additional user requests:** Dietary restrictions and accessibility requirements were integrated for inclusivity of the app -- reaching a larger audience.
    - The list of dietary restrictions or accessibility requirements to be addressed are printed.
    - Individual accommodations can then be addressed by changing the status so other organizers know about it.
- **Mandatory extensions:** Events can be cancelled by the organizers who created the event, once an event is cancelled, all attendees and speakers for the event receive a message. Organizers can create any type of new User accounts.

DESIGN PATTERNS:

- Implicit Observer within MVP
    - The observer being the Presenter and the observant being the View. Although this is not a direct implementation of the Observer design pattern, the MVP inherently has Observer logic; any changes or input read from the user interface affects the presenter. Classes: every single View and Presenter class.
- A modified version of the simple factory for event creation
    - At first, it was considered fully implementing the Factory design pattern. The initial idea was to make an Event interface that could then be implemented by the classes Talk, PanelDiscussion and Party. However much of the code from Phase 1 was written considering that an Event is a class, not an interface. Converting it to an interface would force a refactoring of the entire code so that any Event created is of type Talk, PanelDiscussion or Party rather than of type Event. This

seemed like the optimal route to take as it would take a lot of time to refactor something that isn't broken as is. It was decided to still make an abstract EventCreator class, which would be extended by the classes TalkCreator, PanelDiscussionCreator and PartyCreator as the design pattern states. Essentially, the Factory design pattern has been implemented, but instead of Event being an interface, it was left as a non-abstract class and entities Talk, PanelDiscussion and Party that extend and override methods if necessary were made.

- Implicit Facade implementation within MVP
    - The view essentially acts as a facade, simplifying client interactions and encapsulating how events and users are created, messages are sent, etc. Classes: every single View class.

REASONS WHY DESIGN PATTERNS WEREN'T CHOSEN

- Simple Factory pattern for User creation
  The User creation system was already developed quite efficiently. Attendees, Organizers and Speakers all inherited from the User class. All that was really needed to do was add one more subclass called VIP. A new attribute was added to the Events that decided whether an Event was a VIP event or not. The new changes were implemented with already existing methods. Potentially the Factory pattern could've again been used similar to Event creation, but in this case there already was a template on which to build upon for the Inheritance of the Users and the creation of Users was all handled in one method instead of making separate user creation classes like in task 1. Implementing the Factory pattern would've resulted in a big refactor that wasn't necessary, so the already existing methods were extended to accommodate for the VIP class.

- Adapter Design pattern for Event creation
  Before the multiple Events extension, all the events took in a single int as a parameter to refer to speakerID. Since the new extensions required to be able to have several speakers, there was a plan to implement the Adapter Design Pattern to adapt input from multiple speakerIDs into one. However the Adapter wouldn't have helped much since what was necessary was a change of every single int speakerID parameter to a list of ints. An Adapter wouldn't have helped in situations with 2 or more speakers so it wasn't implemented.

- Strategy in event sorter
  Initially, we thought about implementing a strategy design pattern for the EventSorter class (that handles sorting events by time, title or speaker). However, the implementation was short enough that we did not need subclasses implementing different algorithms. That would require instantiation of a new strategy class every time the user wants to sort events which we thought was overkill for our case, so we just implemented a method that calls other methods based on an input.

IMPROVEMENTS ON PHASE 1:

- Refactoring EventSorter:
    - Refactored EventSorter from controllers to Usecases as that's where it truly belonged (it dealt with user entities). Made the code follow SOLID principles
- Refactored IO Gateways
    - Added an abstract IO class to remove redundancies between MessageIO, EventIO, UserIO.
- AuthManager for better clean architecture
    - Originally we had user interfaces directly interact with an entity, which completely breaks clean architecture, however we came to the solution of creating a use case whose purpose is interacting with the presenter's which in turn interacted with the user interface.
- Refactored controllers that were directly interacting with Entities by handing over some functionality to use cases and keeping track of IDs over the actual entity objects.
- Refactored code to replace Stringbuilders with String objects