Phase 1 - Design Document
Team Variable

Updated Specification:

- Running the application initially asks the user for an existing log-in, or to create a new account using a username and password of their choosing
- Once logged in, a new user can now either:
    - choose to import a compatible calendar file,
    - manually schedule their events to a new calendar
- If the user chooses to manually schedule their events, they will be asked for:
    - the courses/recurring events they are currently participating in
    - the times and days when these events occur
- Once the user has completed the above steps, they can now:
    - View the scheduled events in their calendar
    - Add or remove events (either recurring or one-off) from their schedule
    - View which study group they are in
    - Join a new study group (via a group ID)
    - Create a new study group
- After a user has joined a study group with more than one member, they will be able to access the time in which all members within the group are free to conduct group work/study
- **Data is saved automatically upon logout. And if there is any saved data present, the program retrieves the saved data automatically upon program startup.**
- **Users will have a task list, from which they can create, add, and remove tasks from.**
- **Users can filter their task list for open tasks and closed tasks. Users will also be able to calculate the average amount of time it takes to close tasks.**

Project Design:

1. Adhering to SOLID Principles:

   We believe that our project in general adheres to the Single-Responsibility Principle well, as classes are well defined and each have their own job. For example, the entity classes are all separate and clear, and there is a manager class for each with a Factory class for EventInterface, a controller classes for each except EventInterface, and each GUI classes represent one single window for the program. The one time the classes need to be changed is if the functionality is changed, which is ideal for the Single-Responsibility Principle. However, looking at all the other SOLID principles, we are unclear about how well the program adheres to them. In the GUI classes, we note that there is a way to organize all the controller instances. Each window, except the FreeCalendarFrame, FreeMonthPanel, FreeViewDay, which extend CalendarFrame, MonthPanel, ViewDay respectively, initializes these instances of controller classes themselves, rather than from a superclass, and there are no interfaces that the GUI classes can implement, which is usually a preferred feature for the Open-Closed Principle. For the Liskov-Substitution principle, the same can be applied. If there were interfaces or inheritance within the GUI classes, this principle can be adhered to. We will definitely look to improve on this for a clearer and more well-structured program. The Interface Segregation Principle may not apply very well for the GUI classes, as every window and display is clear in the program and is some feature that the client is looking for. The Dependency Inversion Principle also goes back to the problem of using interfaces in the GUI classes. If the higher level module should depend on abstractions, then we have not adhered to that. While we in theory know what can be improved on within our project in due time, we are looking forward to the feedback from Phase 1 in order to have a clear direction on our next steps.

2. Major Design Decisions:

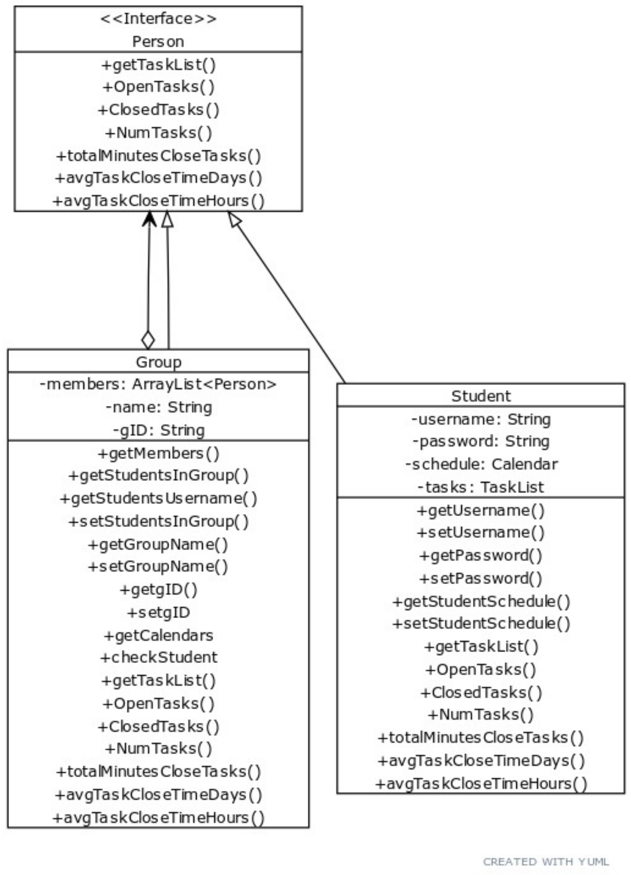
   - Person Interface
   - Event Interface
   - Format of OneOffEvent date (mm.dd)

3. Design Patterns:

a. **<u>Composite Design Pattern:</u>**

Our project has a Student class and a Group class. The Group class can contain instances of Student as well as instances of Group. This allows Group to contain other nested Groups. This structure can be represented as a Tree-like structure where Student can be represented as a leaf and Group can be represented as a node. Since Student and Group can be represented as a Tree-like structure, we are able to make use of the **Composite Design Pattern**.

To represent a single unit we will use the Student class as our Leaf class. And we will use the Group class as our Composite Class since Group can store both Student and Group classes. We implemented an Interface called Person, this Interface contains methods that are common to both Student and Group and we will use the Person Interface as our Component class. These common methods are mostly related to the new Task List feature we implemented in Phase1. And finally, we implemented this Interface into both the Student class and Group class. The following is a UML class diagram to further illustrate this relationship:



b. **Template Method Design Pattern:**

The **Template Method** design pattern defines the skeleton of an algorithm as a base class that contains the standard steps across a set of classes. This abstract class contains the invariant pieces of the domain's architecture, providing a structure with either default implementation or no implementation at all. The abstract class allows its subclasses to redefine certain steps of the algorithm and specify different implementations if needed. In a way, the base class represents a "placeholder" of the algorithm, and the subclasses implement the "placeholder". Furthermore, the Template Method uses inheritance to vary part of an algorithm and modifies the logic of an entire class.

The goal of our project is to find a time slot that works for everyone across a group given their individual schedules. In our frontend, we want to have windows to show the calendar and days for viewing. For example, we have FreeViewDay to see the days people are free and ViewDay, which is a parent class of FreeViewDay. The **Template Method** design pattern is applicable here with ViewDay being the base class and FreeViewDay being its subclass. We have standard functions in ViewDay such as setTable(), getCalendarController(), getDate(). These functions are also applicable and are inherited in the FreeViewDay class. However, there needs to be alternations of steps in FreeViewDay as we have to populate it differently, which leads to an override of the function populate().

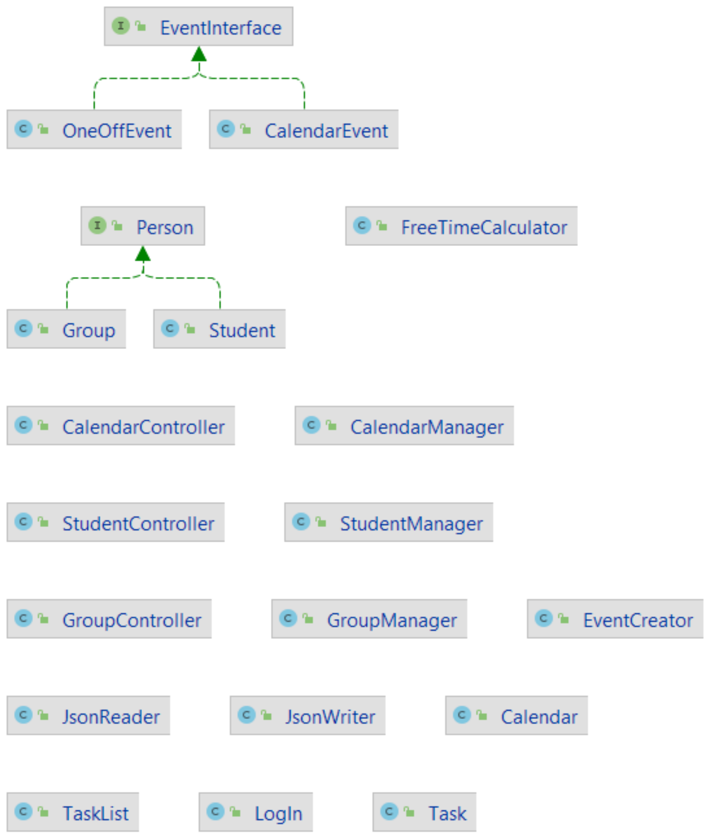
c. **Future Prospective Design Patterns:**

With regards to Phase 2 and future Design Patterns we plan on implementing, we look forward to refactoring EventCreator using the **Factory Design Pattern** to create an EventFactory class. In addition to this, we will consider the possibility of the use of a **Builder Design Pattern** for Student to create a StudentBuilder class, and if we consider year-after-year development, we may consider the need for the **Adapter Design Pattern** in the case of EventInterface or the Person Interface to ensure that future different types of Events and Persons can be accounted for.

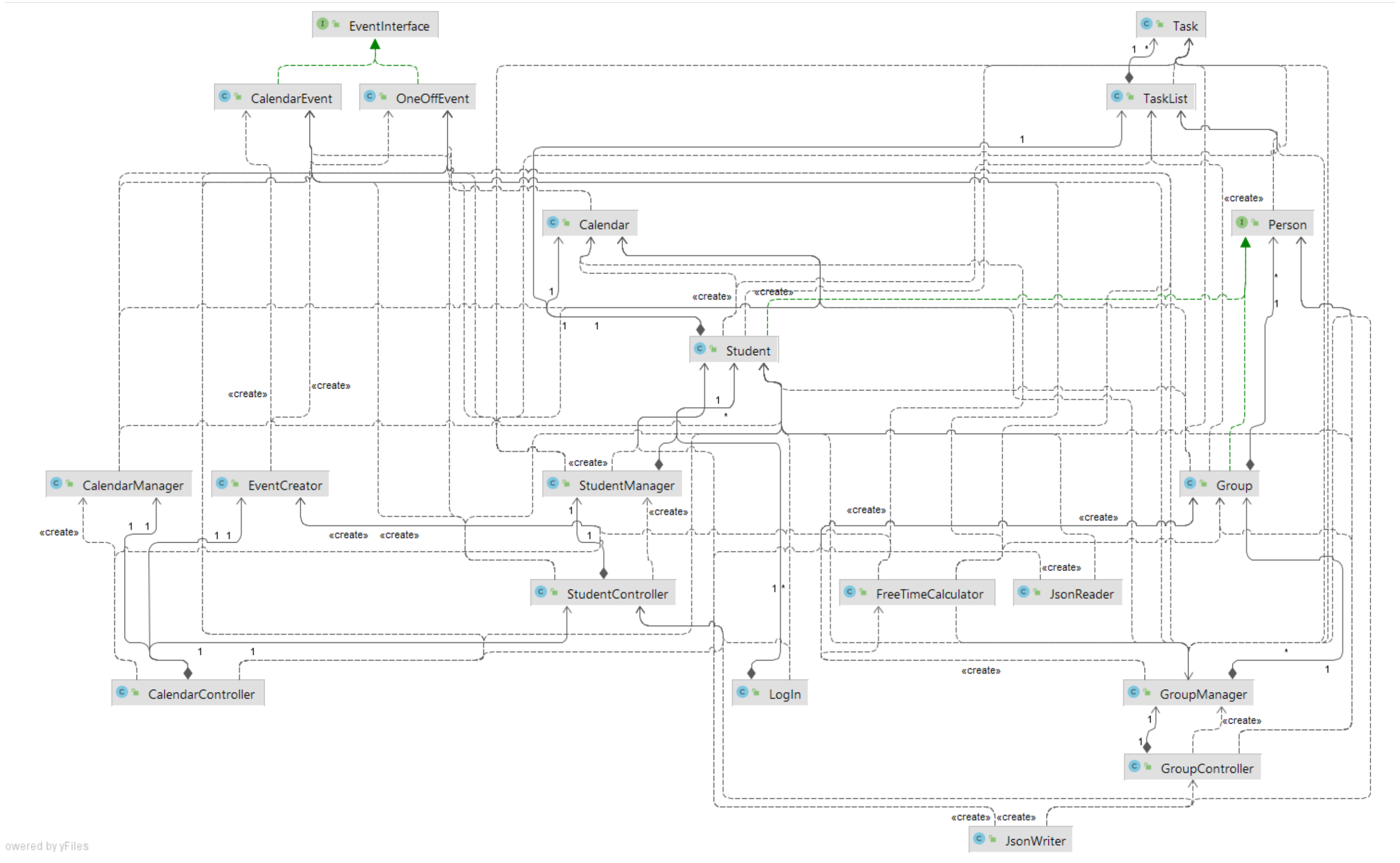
Project Structure/Architecture:

1. UML Class Diagrams:

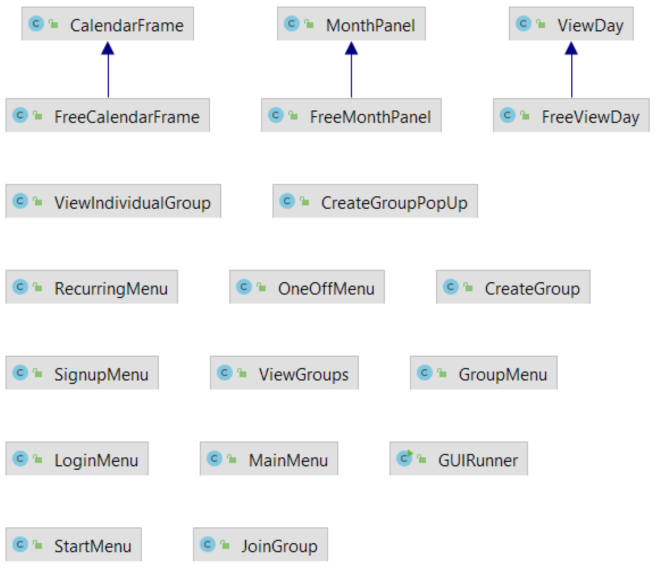
All UML Diagrams can be found in the directory *...phase1/UML Diagrams.* The following is a screen capture of the UML Diagram generated in IntelliJ for our backend classes without dependencies:



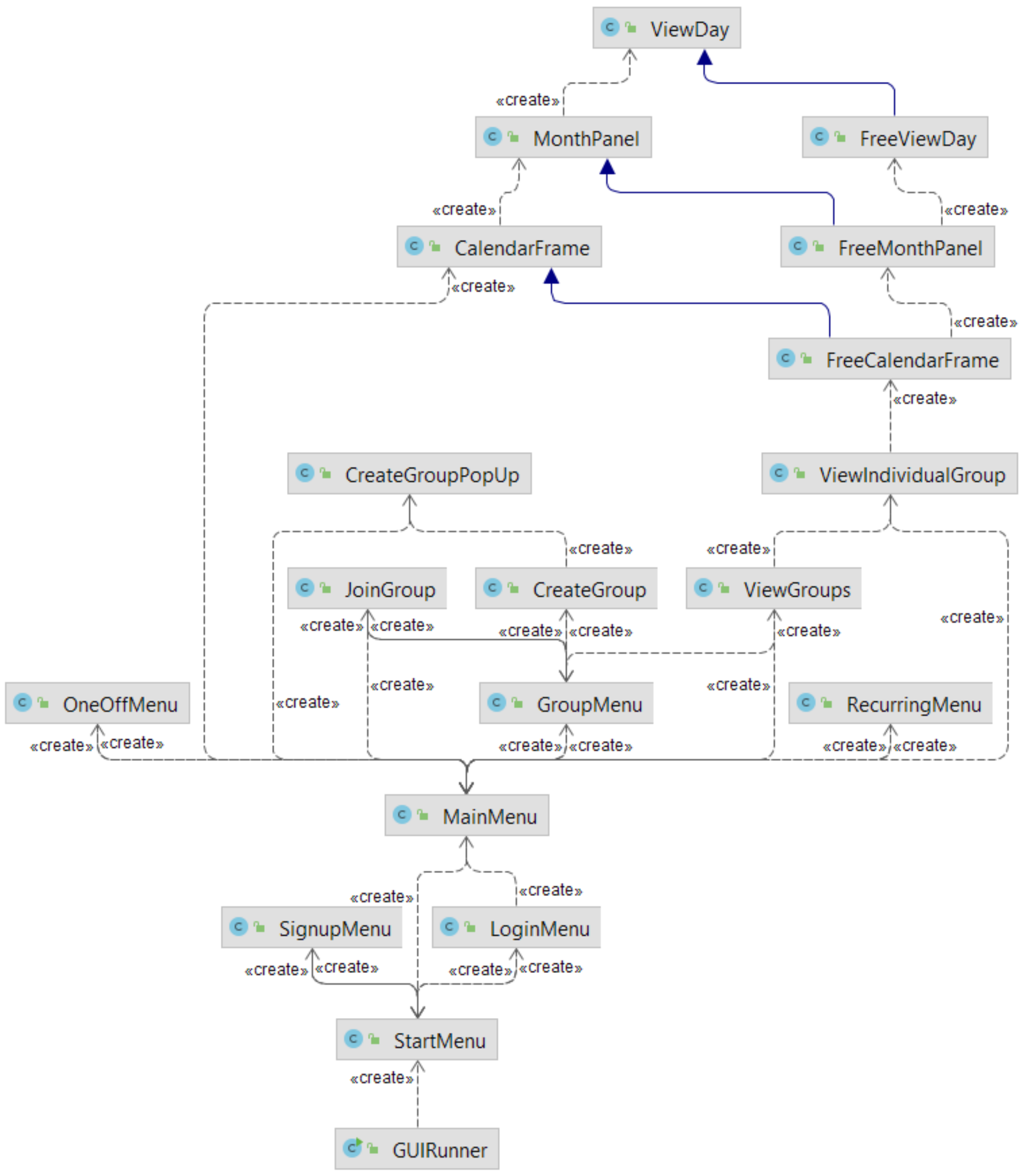
The following is a screen capture of the UML Diagram generated in IntelliJ for our backend classes with dependencies:



The following is a screen capture of the UML Diagram generated in IntelliJ for our frontend classes without dependencies:



The following is a screen capture of the UML Diagram generated in IntelliJ for our frontend classes with dependencies:



2. Adhering to Clean Architecture:

We believe our program adheres to Clean Architecture quite well. To adhere to Clean Architecture, we purposefully created Controller classes wherever and whenever we needed to access certain properties of entity classes within our outermost layer. However, we are also concerned with this design as in theory, all we are doing is repeating the same methods within classes, just with different names at the Manager and Controller level such that our outermost layer only interacts with Controller level classes. This could potentially be seen as *Repeated Code* (see *Code Smell* for more detail about this). Each necessary method that the GUI calls on will always call on it from the instances of the controller classes, furthermore no manager classes are called upon within the outermost layer. An example of the Dependency Rule being consistently followed in the GUI would be in the RecurringMenu and OneoffMenu, where we would add a new event to the current user's Calendar. CalendarController's createRecEvent() or createOneOffEvent() method is called upon respectively, which then calls the CalendarManager's respective methods, which then operates on the entities. Another example would be a helper method called populate() in ViewDay, which helps structure and create a parameter needed to display the single day schedule. This method calls on the StudentController's getTimes() method, given a String of the current username which we have passed in as a parameter, the date, and the day of the week. This method then runs and calls upon the corresponding StudentManager method which then directly accesses the student's Calendar.

3. Packaging Structure

For our packaging structure, we mainly focused on dividing up parts of the program (both frontend and backend) into ways that would make it easier to navigate for all of us. We created packages that were appropriately named, but not over packaged such that it would be difficult to navigate when looking for a specific class. As such, we chose the packages (all in ...src/main/java) ***backend*** (responsible for dealing with solely backend related classes)*,* ***frontend*** (containing classes relating exclusively to the frontend)*,* ***calendar*** (containing classes related to any user's calendar)*,* ***events*** (containing classes related to all user events)*,* ***login*** (containing classes relating to user login and sign up)*,* ***users*** (containing classes relating to individual or groups of users).

Program Specifics:

1. Functionality:

At the end of Phase 0, our team had completed the majority of the backend development needed for our program, having our Command Line Interface run most of the functionality of the original specification. Therefore, our focus for

Phase 1 was ensuring the completion of our backend development, while half of our team worked on the frontend development. Additionally, to save and load information for our program we implemented the classes JsonWriter and JsonReader. These classes leverage the Gson and Jackson libraries to serialize and deserialize our program's data. The JsonWriter class is then able to take the serialized data and write them to a JSON file for storage. As well, the JsonReader class is able to read the saved JSON file and deserialize the data back into our program.

2. Code Style & Documentation:

Over the course of Phase 1, the frontend team has consistently ensured that their code was appropriately documented using JavaDoc, using appropriate method and class names to ensure our code would be easily readable. Additionally, we made sure to backtrack and also documented any source code from Phase 0 that had previously been left undocumented, and that class names and methods were consistent so that all of our code was both readable and easily understood via the documentation.

3. Testing/Refactoring:

a. Testing

Moving on from Phase 0, we tried to fully populate our backend's test cases as much as possible, taking care of edge cases wherever appropriate and also including the test cases for exceptions. For our frontend/GUI, we realise that writing unit tests is difficult and labour-intensive, so we tested our GUI by covering any sequences of operations we could think of, and comparing whether we obtained the desired result.

b. Refactoring

Primarily, all refactors within our code occurred when we dealt with the source code for the backend. An example of this is when we refactored our code after implementing EventInterface as CalendarEvent and OneOffEvent no longer had a composite relationship and both had to implement EventInterface (Pull Request #7 on the main branch).

Additionally, we also refactored our code after the introduction of the Person Interface, as both Student and Group classes implement this Interface (Commit #82be76f on 14/11)

c. Code Smells

When looking at potential Code Smells within our code, we can analyse a case of potentially *Repeated Code.* This would be within the Populate method of the FreeDayView and DayView classes. The reason for this potentially *Repeated Code* is due to the different parameters and timetables we are using within the method. We could potentially also view many of our classes to obey Clean Architecture as *Lazy Classes* since these classes serve no other purpose and are often repetitive in nature, except for allowing the next outer layer a means of interaction without violating Clean Architecture.

4. Use of GitHub Features:

With regards to GitHub features, our team has made use of Pull Requests wherever appropriate, always letting the others know before making any significant changes to the main branch. Furthermore, we explored the Issues feature within GitHub to address major issues when merging or running code, such as whether we should have a Build Configuration for the project given the various dependencies within our code. Furthermore, when handling any debugging within the team, we made efforts to specifically point out Actions that resulted in said error to give context when addressing the issue within our code.

Progress Report:

1. Open Questions:

- What is the expected standard for Phase 2? Are we meant to further implement more ideas or simply amend our project according to feedback and focus on the presentation?
- What differentiates a major violation of SOLID/Clean Architecture from a minor violation?

2. <u>Advantages of Current Design:</u>

The advantages of our current design include (but are not limited to):

- A straightforward means of adding new types of Event due to our EventInterface
- A straightforward means of adding new types of Persons due to our PersonInterface
- HashMaps used in appropriate places to improve efficiency over ArrayLists (considering we are primarily focused with accessing specific indexes)
- A GUI that is very flexible due to our use of Swing as Swing components follow the Model-View-Controller paradigm, in addition to the ability to include 'extras' (icons, etc.) for different components

3. <u>Individual Contributions:</u>

<u>Name of Individual</u>	<u>Contributions Made as of End of Phase 1</u>	<u>Planned Future Contributions</u>
Cathlyn	● Template design pattern ● Unit testing for classes ○ CalendarEvent ○ OneOffEvent - updated class too ○ Calendar ○ CalendarManager ○ CalendarController	
Gilbert	● Using Java Swing for front end development ○ Created `FreeViewDay`, `FreeCalendarFrame`, and `FreeMonthPanel` and connected with backend ○ Help creating `ViewDay`, `CalendarFrame`, and `MonthPanel`	● Help polish frontend GUI display ● Cleaning up the program as a whole by implementing interfaces or using design patterns to fit the SOLID principles and clean architecture if needed. ● Help with implementing importing of iCal files

	○ together with frontend team ○ Created `RecurringMenu`, `OneOffMenu`, and connected with backend ○ Helped creating `LoginMenu` and `SignupMenu` while connecting to backend ● Back End Changes ○ Completed `FreeTimeCalculator` with intended functionality ○ Updated `GroupManager`, `GroupController` for use in front end ○ Updated `CalendarController` ○ Helped updating `StudentManager` and `StudentController`	
Raaghav	● Front End Development using Java Swing ○ Helped design entire layout and structure of FrontEnd code ○ Created Group Menu and added various functionalities by connecting to the backend classes ○ Created ViewDay class and connected it with backend ○ Created ViewGroup class and connected it with backend ○ Adapted CalendarFrame and MonthPanel code from online resources to fit our project's needs ○ Helped develop MainMenu class ○ Helped develop LoginMenu, SignupMenu	● Add upload functionality so that users can drop-in a iCal file, and their calendar can be automatically created ● Making tweaks to front-end design

	classes while also connecting these classes to backend code ● Back End Development ○ Updated Group, GroupManager and GroupController classes for ease of use in Frontend ○ Made a few edits to StudentController and StudentManager	
Rashid		
Tajwaar	Contributions included, but is not limited to: ● Writing an `EventInterface` Interface and refactoring our code appropriately (Commit #`702f4b3`) ● Rewriting our `StudentManager` class to include a `checkValidStudent` method to better utilise throughout the code and for Exceptions ● Rewriting our `GroupManager` class to add a `getGroup` method ● Writing both `CalendarController` and `StudentController` ● Unit testing classes with Cathlyn to ensure our test base was sufficient and well populated ● Writing `Exceptions` ● Lead writer and contributor for Phase 1's Design Document	● Further implement more Design Patterns (such as EventFactory and possible StudentBuilder) ● Further populate test cases as needed ● Aid in frontend development, specifically uploading and handling iCal files ● Further handle and tweak Exceptions

Vergil	● Implemented Task List feature by writing classes TaskList and Task and updating the Student and Group classes. ● Implemented a new Person Interface and used that new interface to implement the Composition Design Pattern for Student, Group, and Person. ● Implemented classes JsonReader and JsonWriter so that the program is able to serialize and save program data to a JSON file as well as read the generated JSON file and deserialize the information back into the program. ● Implemented the saving of data (JsonReader and JsonWriter) into the GUI.	● Instead of saving program data to JSON files, will implement an embedded database instead. ● Implement filtering of Task List by date range. ● For all Task List methods, want to make it work for a specified date range instead of just on the entire Task List.