Note: If our program isn't running, you may have to restart IntelliJ. The way we set up Gradle means it doesn't always play nice with IntelliJ.

Updated Specification

Our program is a scheduling and social application targeted towards University of Toronto students. Students can create an account for themself, which has a unique associated username and password. Once logged in, they are able to input their own schedule and add friends. They can add events to their own schedule, specifying whether an event is for a class, for a social event, a fitness event, or other academic events. This schedule gets saved so that a user can later log in on a different device and still access their schedule.

The program will show the user's schedule on a calendar on the app's main page.
Users can send friend requests to other users, and can accept or decline incoming friend requests from others. Once friends with someone, their schedules can be "shared" with each other. By sharing their schedules, users can see what times others are not available, in order to help them more easily figure out when they can meet with each other. The compare schedules screen shows only colour blocks showing unavailable times, so that both user's have privacy regarding their scheduled events.

Since Phase 1, we have been focusing on continuing to develop the GUI and the rest of our code, so that they are able to interact with each other. The GUI has been fully integrated with our code, so that the user is able to access and properly use all features through the GUI. Due to our original GUI having very large methods, we have since divided the GUI into two separate classes: ScheduleGraphics and ScheduleDrawer.

Our program now includes a log in system where the user can create their own username and password. Once their account is created, any events they add are stored in the database and can be accessed if they log in from another device. Other features include adding different types of events, allowing users to send and manage friend requests, and allowing users to only compare schedules of friended users.

Major Design Decisions

Our decision to create a local application was mainly due to the fact that we had already begun researching information regarding GUIs, and switching to a web app meant we would not be able to use the data gathered about GUIs. Our first version of the GUI can be seen in pull request #12, though it was frequently updated throughout phase 2.

We decided to make users create their own account so that they would be able to interact with other users on the platform, giving our program a social aspect. This would allow them to share their schedules with each other and compare availability. We implemented this feature in pull requests #7 and #8, through the addition of account creator and account manager.

The inclusion of a friends list was due to privacy concerns. Without it, there would not be a way for users to select who could see their schedule. As a result, creating a schedule to share on the program would make it accessible for everyone. The friends list allows users to give permission to trusted individuals to view their schedule. The add friends feature was created in pull request #9.

We opted to use a database so that we would be able to save the information of a user between different runs of the program. Additionally, the share feature required that a user needed to see another user's schedule, which means our program must save the information of other users as well. This decision can be observed in pull request #13, where we connected our program to a Google Firestore database. The decision to use Firestore was due to it being free for users who do not require frequent reading from the database, its simple use and set up, as well as the fact that we had done research on it beforehand.

Packaging Strategy

For our code's packaging strategy in phase 1, we chose to organize our files based on clean architecture structure, matching our CRC model. A folder is designated to each level of clean architecture: Entity Classes (another folder is within this for Events), Use Case Classes, Controller, and Interfaces. Each file corresponds with a section of our CRC model, as illustrated in our UML diagram. By using this packaging strategy, our organization stays consistent through all aspects of our project and is easy to navigate and maintain. In phase 2, our program structure did not undergo any drastic changes, so we kept our previous packaging strategy.

SOLID Principles

Our code adheres with the SOLID design principles in order to make our software creation and running be as smooth as possible.

In order to follow the single responsibility principle, each of our classes is responsible for a single task or requirement, with attributes critical to its functionality.

Our code follows the open-closed principle with the structure of our classes interacting with each other. No class is able to be modified from the outside, however they can be extended

and have their method called by other classes, so that our code can interact with all classes when needed.

All instances of Event can be substituted by the appropriate Event subclasses and not affect the correctness of the program, consistent with the Liskov substitution principle.

Since Phase 1, we have implemented a user interface and therefore we needed to make sure to follow the interface segregation principle. For instance the user is given multiple buttons at the top of the GUI, that they can then select what tasks they wish to do. By keeping these features separate, both visually as well as in our code organization, it means that features are not called on unless needed.

By following the layers of clean architecture, our code abides the dependency inversion principle. Based on its placement within our code structure, all classes have clear dependencies and follow them throughout the code.

Clean Architecture

The Layers

Our program is segregated by the four layers of clean architecture.
We currently have four entity classes: Person, Events, Schedule and Reviews.
Our use case classes, UserList, ScheduleEditor, ScheduleComparer, FriendAdder, AccountLogin, AccountCreator and AccountEditor use our entity classes to carry out our program functions.
Our controller classes ScheduleManager, AccountManager, InformationSaver and CourseDataGetter then call on methods from the use case classes to enable the interface class to use the program.
Finally our GUI, ScheduleGraphics works as the interface that calls on our controller class and methods for the client and we have created 2 more classes ScheduleDrawing and Compare Graphics that are used in scheduleGraphics in our frameworks and drivers.

Changes

Originally, our graphical user interfaces worked independently from the other classes in our program, but for phase two we have implemented calls to our controller classes AccountManager and ScheduleManager and cleaned up all calls to and instances of use case and entity classes. An example of this is that we needed to use an instance of the entity schedule in our GUI and to ensure we did not break the layers of clean architecture to access it, we created a use case class UserList. Thereby adhering to clean architecture.

We have carried out refactoring to clean up instances of skipping layers of clean architecture in multiple classes such as ScheduleManager and ScheduleGraphics and ScheduleDrawing and account manager, and our code now only calls on methods that are one layer below so as to not call methods from layers above or more than one layer below.

Scenario Walk-through

A scenario walkthrough of our program is as follows: After logging in to our program, a user will add or edit an event to their schedule by clicking the respective button. This uses the actionPerformed method under the relevant actionListener class as a response to this button being clicked. The actionListener methods call on ScheduleManager to add or edit the event. ScheduleManager is an interface adapter that calls on the required use case class according to the method. In the case of addEvent and editEvent, it calls on the use case class ScheduleEditor which in turn changes the value of the user's schedule using setter methods in the entity class or instantiates an event in their schedule. Once the change is carried out, our GUI , ScheduleGraphics, calls on AccountManager that calls UserList to iterate through a user's schedule and represent their events on the interface. A similar process abiding by Clean Architecture exists for comparing schedules.

We have decided to provide a link to a video demonstrating our program's functionality as we go through the scenario walk-through described above: https://drive.google.com/file/d/1L__uMMlqkYPNkgjC4AUOCa9pAkleZ4T6/view?usp=sharing (Note: There is no sound.)

Design Patterns

EventFactory (Simple Factory)

We applied the creational Simple Factory design pattern to the Event class, which is an entity class of our program that represents events in a person's schedule. This implementation can be seen through pull request #14 "events factory package" (https://github.com/CSC207-UofT/course-project-group-043/pull/14).

We created an EventFactory class that would be solely in charge of creating the objects of different types of events: CourseEvent, AcademicEvent, SocialEvent and FitnessEvent. Within EventFactory we have conditionals that decide (based on given string input of the type of event that the user wants to create) which of the event subclasses to instantiate for this person's schedule. EventFactory is called and provided inputs by the addEvent

method in ScheduleEditor. ScheduleEditor gets called by our controller class, ScheduleManager, which receives the user's inputs from our GUI.

For now, we have only four types of events, but depending on whether we want to add more types of events, it would eventually cause the conditionals and overall method code in the createEvent method in EventFactory to be quite long. Thus to prevent this code smell later, we could alter this simple factory design to a factory method design.

ScheduleManager, AccountManager (Facade)

In addition, our ScheduleManager and AccountManager serves as a facade, which is a structural design pattern. As previously mentioned ScheduleManager is a controller class that represents and manages the entire system of users and schedules.

A user might want to add, remove, or edit an event. Since the user can now also have friends, a user might want to compare their schedules with their friends and send/accept friend requests etc. Altogether these actions require a large and complex framework, which is why we create a facade class like ScheduleManager to hide the framework's complexity behind ScheduleManager's simple interface.

Thus, in the end, our graphical user interface won't have to depend on the many classes from the complex framework and if we ever want to change to a different set of frameworks we can just make those changes directly in the facade class instead of throughout the user interface.

AccountManager is another controller class that represents and manages the different accounts associated with different users including the log-in info. In the same way, this facade allows the user interface to easily change different aspects of an account without having to mess with the detailed classes involved. This can be seen in the following pull request https://github.com/CSC207-UofT/course-project-group-043/pull/8.

Plans For The Future

These are the design patterns we had time to successfully implement thus far. However, if we had more time to further develop our program there are different areas where other design patterns could be implemented.

For example, with more time we could have linked our reviews of locations to the creation of a particular event. This entails that we would have reviews for different types of locations such as for working cafes, restaurants, bars, gyms or libraries. Evidently,

each of these locations pertains itself to a particular type of activity or event. For instance, one would definitely create a FitnessEvent to go to the gym rather than an AcademicEvent. Firstly, the separation of these different types of reviews can be represented using a factory design pattern, similar to that of our event entities. Secondly, linking the type of event created to reviews of a location could potentially be done using an observable design pattern. Imagine that someone wants to create a FitnessEvent on their calendar to do some exercise. With this implementation we could have the EventFactory class implement a notification method that would call a method in the new ReviewFactory to produce a list of reviews for a specific type of location according to the event for the same user.

Use of GitHub Features

For phase 2, our group made better use of two GitHub features, namely the GitHub project task tracker and the pull requests. The utilization of these two features greatly helped our coordination in phase 2 compared to the other phases.

The GitHub project task tracker was more utilized to manage our current project tasks. We used this to keep track of tasks we needed to complete for the project in the “to do” tab, made sure to check there regularly and moved the task to the “in progress” tab when one of us was doing it. This allowed for better communication and we were able to easily allocate different responsibilities for different team members.

We also better utilized pull requests to improve communication by assigning reviewers to look through the code and approve/request changes for them. We could also leave insightful comments that could help us identify potential mistakes in our code. This was helpful in case there were errors in the code that weren't initially noticed.

Code Style, Documentation, & Code Organization

Every class is documented, though some methods have not been because we decided the headers for them are self-explanatory and adding comments would be redundant. As discussed above, packaging helped us keep our code files organized, although some files themselves turned out very long, resulting a bloater code smells. (Examples: ScheduleGraphics, ScheduleDrawing). Furthermore, the addEvent, removeEvent, and editEvent methods have long parameter lists, another code smell. However, we ended up with these when trying to fix Clean Architecture violations and remove uses of the Person class from Controllers AccountManager and ScheduleManager. While we believe the

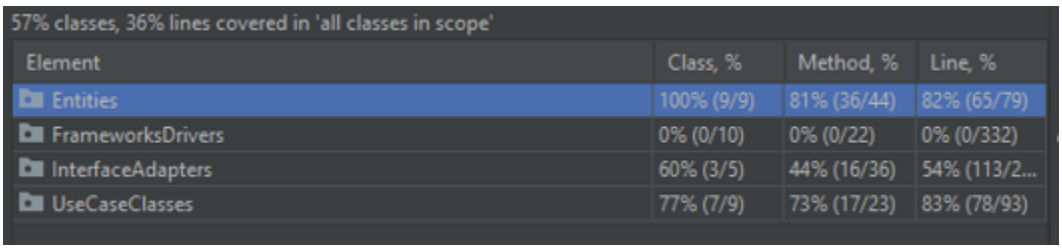
documentation should clarify what each class and method is meant to do, with more time we would have liked to separate out our larger classes into smaller, more logical components.

Testing & Refactoring

From Phase 1, we had a violation of clean architecture because some of our interface adapters were storing entity objects with them. Our solution to this problem was to create a use case class which stored the entities, and then have our manager store the use case class. In pull request #21, we solved this issue for AccountManager, and in pull request #27, we did the same for ScheduleManager.

Due to our lack of a GUI at the end of phase 1, we stored our events in a Schedule object, which was a hashmap that held the days of the week as a key, and a hashmap as a value, which had the time of day as a key, and name of the event taking place as a value. However, this representation would not work for our GUI, as we needed to have access to the specific event objects on the schedule, rather than just a string representation for every hour. In pull request #28, we made this change to allow for Schedule Objects to work with our GUI.

Since the end of Phase 1, we have added more tests to our program. When running the tests, the coverage is shown below:



Progress Report

Name	**Completed Work**	**Specific Pull Request Demonstrating Contribution**
Dennis	Worked on specific GUI components such as comparing schedules, and logging in. Connected our database to the rest of the program.	Pull request #31: “Dennis g UI” Demonstrates creation of an entire log in screen and integrating it successfully with the rest of the program — integral to even starting using the program
Emma	Continued to create and form GUI, and make it interact with the other code.	Pull request #12: “Uploading work in progress ScheduleGraphics”

	Helped make add event take it's input and display an event on the screen, as well as creating the manage friends button in order to manage and send friend requests, and view current friends.	Shows the initial creation of a graphical user interface, laying down the foundational components and beginning to lay groundwork for adding other features later on
Priyanka	Changed implementation for creating events to work with EventFactory, changed the format of our schedule so that it would be more efficient for the user interface, refactored various classes to work better with the new form of Event and its subclasses, created tests for ScheduleManager and Event.	Pull request #14: "events factory package" Shows implementing the Simple Factory design pattern and creating the various types of Events that are used by the program and put into a user's schedule. All other components of the program rely on this.
Rachel	Refactored some controller classes (AccountManager and ScheduleManager), created UserList, implemented Facade design pattern in them, set up gradle files, filled in missing javadocs, made the data getting process more forgiving of incomplete course names	Pull request #21: "Refactored AccountManager to avoid Clean Architecture violation" Shows creating new classes to avoid Controllers skipping a layer to interact with Entities. Having this also helped avoid some Clean Architecture violations in GUI classes.
Tim	Made it so that recommendations were saved in our database and could be accessed between runs of the program. Refactored controller classes and UseCase classes.	Pull request #24: "Tim" Created new UseCase and Controller classes to avoid Clean Architecture violations, and implement the Review class to the database, allowing information relevant to an entirely new feature to be stored between uses of the program (This pull request is marked as closed instead of merged because there were strange issues we couldn't figure out. I directly pushed the changes on this branch to main and it worked.)
Sunehra	Wrote ScheduleDrawing (splitting ScheduleGraphics), expanded on GUI functionality, added ability to add and edit events from the GUI, fixed issues with schedule not displaying properly	Pull request #22: "Sunehra(friend-adder)-27/11(paintevent)" Core part of the program is creating a schedule and comparing it, which isn't possible without adding and editing events. This demonstrates a significant

		contribution to both the GUI and to a key feature of our program (creating schedules).

Accessibility Report

Principle 1: Equitable Use

Our program has the same means of use for all users. Every user is able to keep their schedule private or share it if they wish to do so. This allows users to choose the level of privacy that they are most comfortable with, and either keep their schedule personal or share availability access with approved friends.

Principle 2: Flexibility in Use

The program can be used by anyone who wishes to plan out their week using a schedule. Although the original intent of the program was for personal schedules, an account could also be used by a group or business in order to plan out weekly meetings and events.

Principle 3: Simple and Intuitive Use

Users are presented with simple prompts when pressing buttons, such as when pressing manage friends, or add events. Errors will return an informative message, such as when adding a user who does not exist, or trying to compare schedules with a user that is not on your friends list, the program will return a message stating so. A help button has been added with simple explanations for each button, so that the user can reference it if confused about how to do certain actions.

Principle 4: Perceptible Information

In the future, we could consider adding a way for users to adjust the size of the menus, prompts, and schedule. This would allow a larger or smaller display to accommodate vision limitations. Creating the option of different colours for event buttons would make it easier to differentiate events on a cluttered schedule. Additionally, having a colour customization feature for the overall display of the program would allow users to make their display higher contrast or have a different appearance to aid in visuals.

Principle 5: Tolerance for Error

Since the goal of our program is to create a customizable schedule for the user, we do not have many restrictions on inputs for their event information. If they have made a mistake our would like to make changes, our program gives users the ability to remove or edit events that are accidentally added to their schedules. Additionally, the cancel button allows users who accidentally press the wrong button to go back.

Principle 6: Low Physical Effort

Our program does not require many repetitive actions or physical effort. The need to type has been minimized through the addition of dropdown boxes selecting options. The display and distance between features and buttons is reasonable, resulting in very little mouse movement required from the user.

Principle 7: Size and Space for Approach and Use

Due to our program being an entirely digital form of software, this principle is not relevant to our program. The only use required are described in Principle 6, and rely on the user's computer manufacturer's accessibility for use.

Write a paragraph about who you would market your program towards, if you were to sell or license your program to customers. This could be a specific category such as "students" or more vague, such as "people who like games". Try to give a bit more detail along with the category.

Since our program has data for courses at University of Toronto, our market would mainly be for UofT students, as they would easily find the most use out of the data. However, anyone who wanted to simply keep track and organize their own time could still find our app appealing. If this program were to continue development after this project, it could expand by gathering course data from other local universities in order to appeal to them. Moreover, anyone who would want to compare schedules with others for any purpose would also find great use out of our app.

Write a paragraph about whether or not your program is less likely to be used by certain demographics. For example, a program that converts txt files to files that can be printed by a braille printer are less likely to be used by people who do not read braille.

Due to the nature of our program, it is unlikely that our program is less likely to be used by certain demographics. As a scheduler, It can be used by anyone that wishes to plan out their day or week. The additional features, such as adding friends or comparing schedules, would require other users as well, but these features are not mandatory. It is unlikely that our program would be used by demographics such as younger children, since they do not have a need for organizing their weekly schedule and arranging times. As well, since our program does not currently include accommodation features such as vision assistance or text to speech, it is not accessible to vision impaired individuals. In the future however these features would ideally be added, since many individuals within that demographic may find the program appealing.