Updated Specification

Our program is a scheduling and social application for University of Toronto students. Students can create an account for themself, which has a unique associated username and password. They can then input their own schedule, specifying whether an event is for a class, for a social event, for a fitness event, or for other academic events. This schedule gets saved so that a user can later log in on a different device and still access their schedule. If the user chooses to input a class event, the program can provide a list of lecture sections that the user may choose from to get those lecture times automatically filled in.

The program will show the user's schedule on a calendar on the app's main page. Users can "share" their schedules with others that they are friends with, which means they have sent a friend request to that other user and the other user has accepted it. By sharing their schedules, users can see what times others are not available, to help them more easily figure out when they could meet with each other.

The program will also provide a recommendation feature. Users may leave suggestions of various locations around or near campus that may serve as good places to study, eat, get coffee, etc., and other users can leave reviews on the suitability of these suggestions.

The notable new features that have been added since the end of Phase 0 are as follows:

- GUI
- Different types of events
- Sharing schedules as an act specifically for friends of the user
- Recommendations
- Accounts for users (this had previously been discussed in the specification but not completely implemented)

Note: While the code for all of these features exists, it is not currently accessible to a user who is only interacting with the GUI. Our intention for Phase 2 is to focus more on improving the GUI and "integrating" it more into the more complicated features of our program so a user can fully, properly use these features.

Major Design Decisions & Justifications

A large decision that our group had to decide on was whether we wanted to create a web app or a local app. Both options come with their own share of downsides and benefits. Ultimately, we decided to create a local app, as our group had already begun looking into creating a GUI for our program. Switching to a web app would change the course of our plans, and it is much preferable to work with what we already had.

Additionally, we decided to refactor our Event class. Though it was originally a much simpler class, we figured that since events ranged from academics to extracurriculars, we could add multiple classes that extended it. This would also give us the opportunity to apply our knowledge of design patterns as well. Given that we needed to create similar objects of different types, we were able to decide on the simple factory design pattern.

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. This was mainly due to the fact that our own group had trouble scheduling meetings due to all of the conflicts.

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. Obviously if someone were to log onto their account, they would want to have the same schedule that they created previously. 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.

Adherence to Clean Architecture (See also: UML Diagram in CRC Model Slides)

Our program is segregated by the 4 layers of clean architecture. We currently have 4 entity classes Person, Events, Schedule and Reviews.
Our use case classes, ScheduleEditor, ScheduleComparer, InformationSaver, FriendAdder, AccountCreator and AccountEditor use our entity classes to carry out our program functions. Our controller classes ScheduleManager, AccountManager 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.

Consistency with SOLID Design Principles

Our code adheres with certain SOLID design principles such as the single responsibility principle and the open-closed principle however, due to the lack of interfaces in our program it is not concerned with the interface segregation principle. Each of our entity classes represents a single piece of data in our program with attributes critical to its functionality and each of our use case classes fulfil only one requirement or procedure in our program thus following the single responsibility principle. Our new implementations

of the FriendAdder use case extended the code in the Person class and ScheduleManager to add a function to the program, however it did not alter any primary existing code following the open-closed principle.
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.
Our code is not fully integrated with our GUI as of yet, however we plan to only allow it to interact with our controller classes. ScheduleManager will in turn call use case classes and their methods to fully abide by the dependency inversion principle.

Discussion of Packaging Strategies

For our code's packaging strategy, 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.

Design Patterns Implemented

We applied the Simple Factory design pattern to the Event class, which is an entity class of our program that represents events in a person's schedule.

We previously had it so that ScheduleEditor (one of our use case classes that edits a person's schedule) could simply add and remove general event objects to a person's schedule. However, we realized that there are different types of events and even though they are still all events, each may differ on certain features. So we turned our Event class into an abstract class with four event type classes that extend it: CourseEvent, AcademicEvent, SocialEvent and FitnessEvent.

We then created an EventFactory class that would be solely in charge of creating the objects of different types of events instead of having ScheduleEditor decide on which type of event is being created. 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.

Depending on whether we want to add more type of events eventually we could potentially alter this simple factory design to a factory method design which would help reorganize the code especially since the conditionals in the EventFactory would be quite long.

Progress Report: Open Questions

What are the best practices for handling exceptions? Our class courseDataGetter throws several exceptions, so what can we do to reduce the need to do this in the first place? We were thinking we need to try and let the data getting process allow more possible inputs to make it more user-friendly, so that the program is less likely to throw an exception.

Do we prioritize letting users have more freedom in their inputs, or a safer, more limited design? For example, when users want to leave a review for a location, should they be required to choose from a list of locations or input any location themselves?

The way our current program works requires that whoever wants to run the program downloads a key to allow them to read and write information to the database where we're storing information, which can be a somewhat complicated process. Is there a way to deal with this that doesn't require the user to download a key?

Progress Report: What Has Worked Well

There are several things that have worked well with our design. The classes are separated within the four layers of clean architecture, which ensures that we can easily test our use cases as well as make changes to our outer layers without impacting the inner ones. This also gives us an easy way to package our files. Additionally, our code also satisfies certain SOLID principles, which allows us to make small changes or large refactors without difficulty. This also gives us the option to expand parts of our program without having to rewrite our classes. Finally, our use of the simple factory design pattern means that we easily create different types of events without having to know ahead of time which event needs to be created.

Progress Report: Summary of Each Person's Work & Plans

Name	**Completed Work**	**Future Plans**
Dennis	Set up reading and writing to a database, wrote classes for managing and editing user accounts	Figure out how to send to the database without forcing the user to go through the whole process of downloading and

		setting up a key. Implement a login feature.
Emma	Planned and researched different GUI methods. Wrote code for GUI and began integration with the rest of the program.	Allow user input that takes in information relevant to / necessary for using other features (recommendations, auto-getting lecture times)
Priyanka	Refactored ScheduleManager to work better with new code, researched methods of fetching course information from a webpage, implemented Factory Design pattern for making different types of courses	Making sure creation of various events works with courseDataGetter, incorporating more useful / efficient data types in the code we have. Seeing if we add more event types whether we should implement a Factory Method design pattern instead of a simple factory.
Rachel	Wrote classes for creating user accounts and for fetching course information from a webpage	Make getting data process more user-friendly — more forgiving of different cases / misspellings / incomplete names
Tim	Implemented recommendation feature — user can recommend locations for studying / getting food and leave reviews / ratings	Implement saving the recommendations between runs of the program
Sunehra	Planned and researched for GUI development, added functionality for marking other users as friends, with additional person attributes, planned and wrote FactoryEvents for the Factory Design implementation of Events.	Implement ‘tagging’ locations from the recommendation feature — i.e. specifying that a certain location is being recommended because it’s a good study spot, implementing finalised GUI features such as calendar.