Alexia Monize, Aman Rana, John Park, Mary Ditta, Poorvi Sharma, and Rahul Jaideep
Group-022
CSC207H1F
Monday, December 6, 2021

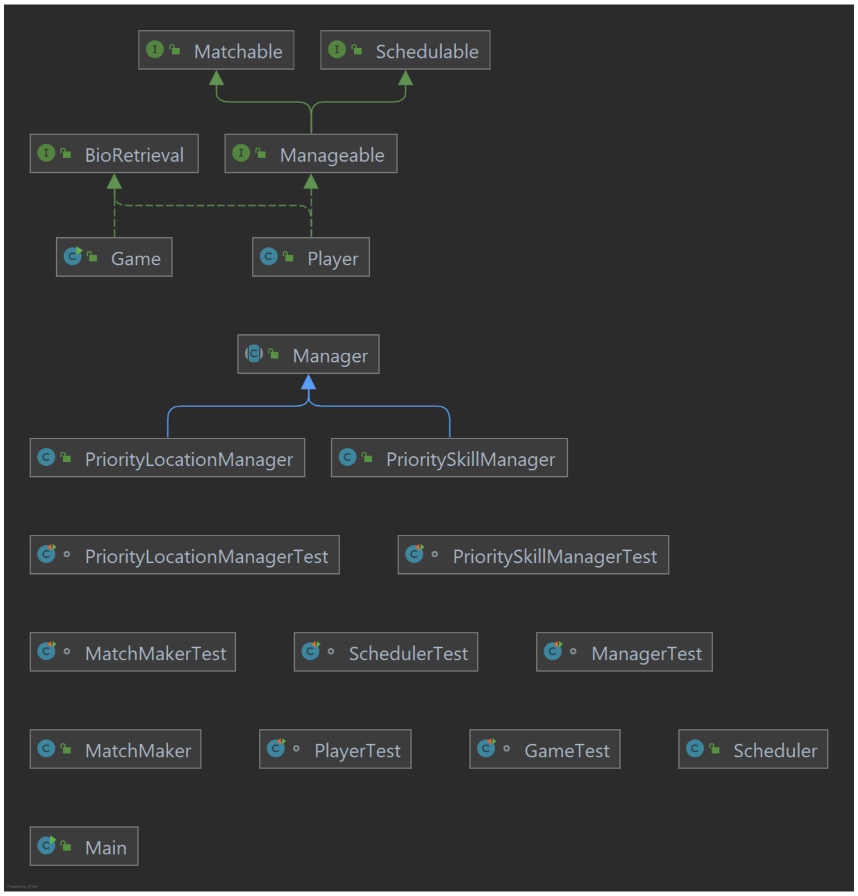
Design Document (Project Phase 2)

Updating Phase 0 and 1

Updated Specification:

- Running the program allows users to swipe through potential tennis players to match with, prioritized by location and/or skill level depending on the user's choice.
- Users have the ability to create profiles detailing personal information, tennis skill levels and availability, that is displayed to other potential matches.
- When a mutual match is made (both users independently choose to match with the other), users have the ability to instant message each other using their individual phone numbers, see information about their overlapping availability, and schedule matches.
- From their profile, users can also see a list of their upcoming matches, their past matches, as well as personal statistics about their wins and losses.

UML:



Major Design Decisions

In Brief:

- Rehauled packaging structure
- Implement Observer Design Pattern
- Developed new classes and interfaces
- Made use of the Pull Request feature on GitHub
- Refactored Code
- Added new classes

GitHub:

- Maintained Issues
- Maintained Projects tab
- Utilized Pull Requests
- Utilized standard Push, Commit, Merge, and Branches functions within GitHub

Design Principles

Design Principle	Violated? (Y/N)	How it keeps to/violates principle (1 example each):
Single responsibility principle	**No - Not Violated :)**	*This principle states that a class should have only one job.* ● The Player class keeps to the principle ○ Calling for Player would issue all information of Player being returned and only has one job, which is to store the data provided by the user
Open-closed principle	**No - Not Violated :)**	*This principle states that entities should be open for extension but closed for modification.* ● The Matchable interface keeps to the principle ○ Matchable is extendable without modifying the class itself, the information from Matchable is used within the Matchmaker class
Liskov substitution principle	**No - Not Violated :)**	*This principle states that every subclass should be substitutable for their parent class.* ● PriorityLocationManager and PrioritySkillManager both implement this through extending to Manager. ○ Both priority options feed data from MatchMaker and Scheduler
Interface segregation principle	**No - Not Violated :)**	*This principle states that a class should not implement an interface it does not use, nor should it depend on methods it does not use either.* ● All classes are implemented such that they do not implement any unnecessary interfaces or depend on random methods. ○ Game implements BioRetrieval

		○ Player implements Manageable, BioRetrieval, and PlayerDataOut
Dependency inversion principle	**No - Not Violated :)**	*This principle states that all entities must depend on abstractions and high-level modules not depend on low-level modules, allowing for decoupling.* ● All the classes are designed in a way such that the entity class is independent of all other classes (Scheduler, MatchMaker, and Manager). ● In this way, their dependency is unidirectional (for example, Player -> MatchMaker/Scheduler -> Manager). As a result, there are no classes that Player depends on, which would be a violation of the dependency inversion principle.

Clean Architecture

Adherence to Clean Architecture standards was a large consideration as we finalized our project. We can outline how exactly we adhered to the principles of Clean Architecture through each level.

Entity Classes:

We have followed the structure for entity classes by having the Player class as an entity class. Our entity classes should represent the critical variables and critical methods within our program. Therefore, within our Player class we also have the appropriate getter and setter methods that will be important for all other parts of our program.

Use Cases:

In addition to our entity class Player, we also have the appropriate Use Case classes as well (i.e., MatchMaker and Scheduler). These adhere to the principles of Clean Architecture because they are making use of our entity class, Player. In addition, we can see exactly how these classes allow us to conduct application specific functions which are described by their names.

Frameworks and Drivers

Most significantly, the transition to a proper frontend using Javascript and HTML instead of the IntelliJ console allowed for a complete separation between our frontend and backend; our UI/web component operates without dependency on the inner layers, and vice versa. The two only communicate through “pipe” JSON files, which follows the clean architecture flow of data.

Interface Adapters

In addition, we can see that we make proper use of Interface Adapters in tandem with our Use Case Classes and Entity Classes. We can see this by taking Schedulable and Matchable as specific examples. These interfaces allow us to interact with the entity class and gain specific attributes that are necessary within our Use Case classes. Therefore, we are not violating the Dependency Rule nor the integrity of the levels policy. We have done this for all

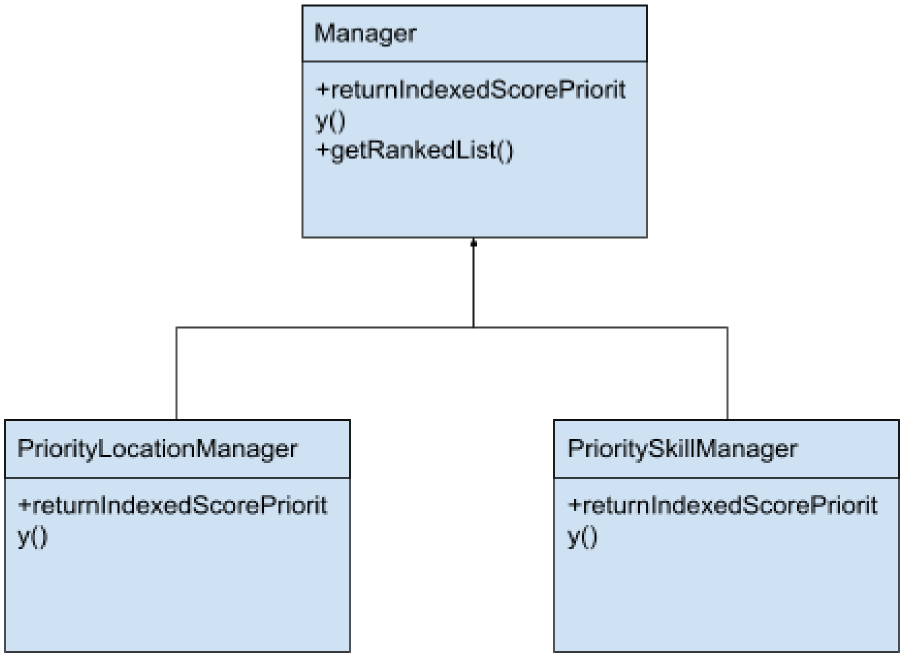
Use Case classes where there are the appropriate interface adapters set in place to accommodate the integrity of both dependency rules as well as policy and level.

Design Patterns

Implementing the Composite Design Pattern

The design patterns that we will be implementing are the composite, template, and adaptor design patterns. For the Composite Design Pattern, we have implemented a getBio() method in both the Game and Player class, but initialized it within the interface BioRetrieval. This exemplifies the Composite Design Pattern as we have a single method, getBio(), retrieving the bio of either a player or game, where the method getBio() has specialized behaviours within the Game and Player classes. In addition, a game is composed of players and exemplifies the appropriate hierarchical structure. In addition, retrieving the bio for a game or a player are two different behaviours: the bio of a game requires parsing a new string that displays the information of the game (i.e. date, time, and players); retrieving the bio of a player relies on an attribute within the player class where our getter method retrieves just the attribute of the bio. They require the same thing, a brief description, however the way they are implemented differ and require the interface BioRetrieval to bring them together.

Implementing the Template Design Pattern


Manager
+returnIndexedScorePriority()
+getRankedList()
PriorityLocationManager
+returnIndexedScorePriority()
PrioritySkillManager
+returnIndexedScorePriority()

The template structure works by having a common main algorithm which contains certain methods which are implemented differently depending on what is asked for. These subclasses redefine those steps without changing the algorithm’s structure. In our implementation the main algorithm is in the Manager class, the steps of our algorithm are: getting scores for each player, signifying the suitability of each player against the user, Using those scores to rank all of the players by suitability, and finally cycling through all of the players and offering them to the user to see if there are any matches. returnIndexedScorePriority is the variant step which is implemented by the derived classes PriorityLocationManager and PrioritySkillManager.

In our implementation, we worked to turn the single existing algorithm into the template structure as outlined above. We made Manager an abstract class and created two subclasses called PriorityLocationManager and PrioritySkillManager. The getRankedList() method in Manager returns an arraylist of manageable items and ranks the list of manageable objects based on their score with user1 (The primary user). The returnIndexedScorePriority() method in the getRankedList() method is the variant method that is implemented differently by PriorityLocationManager and PrioritySkilledManager. These methods return doubles, which represent the suitability of a match between two users. The lower the score, the more compatible the two players are. PriorityLocationManager implements it in such a way that the score is weighted more on location than skill. Whereas, PrioritySkillManager implements the method in such a way that the returned score is weighted more on skill than location.

In Phase 2 we refactored the Manager class to make it more readable and to conform to software design principles.

Implementing the Observer Design Pattern

As we began to think about facilitating communication between our frontend and backend, we realized we would need to implement the Observer design pattern. The observer design pattern keeps track of “observers”, for which it notifies them of any relevant state changes. In our implementation, both the frontend and backend are responsible for reading from and writing to “pipe” JSON files. When either end is in a “waiting” state (ie. the backend is waiting for user input before it can manipulate user data), an function runs that continually checks the relevant file’s metadata to see if the file’s last updated time is more recent than the last time the file was updated and read (ie. if changes have been made to the file). If so, another method is called (our observer) to read information from the file and pass it to the necessary class.

Packaging Strategy

Compared to Phase 1, we have implemented a form of Packaging. Specifically, we have divided up the classes into three categories of features: Managers, MatchingAlgoHelpers, and UserData.

- The Managers package contains Manager, PriorityLocationManager, PrioritySkillManager and the Manageable interface.
- The MatchingAlgoHelpers package contains the MatchMaker and Scheduler classes as well as the Matchable and Schedulable interfaces.
- The UserData package contains the BioRetrieval, Game and Player classes.

We recognized that there were three specific categories of functions or attributes of our program, where each class and their appropriate interfaces could be grouped together. An example of this lies within the Manager package. The Manager class implements the Manageable interface and PrioritySkillManager and PriorityLocationManager are subclasses of

Manager. Within the MatchingAlgoHelpers package there is the MatchMaker class which implements the Matchable interface. Lastly, we have the UserData package where we have the Player, Game, and BioRetrieval classes. These are packaged together because of our Composite design pattern. Both Player and Game classes implement the BioRetrieval method in the BioRetrieval interface. Therefore, because of this composite relationship Player and Game hold, we place them in the same package.

Testing, Refactoring & Code Organization

Our project is thoroughly tested. All our classes are completely tested using JUnit 5, with each method having at least one test case. According to Intellij Unit Testing, after running all the tests, our test coverage was around 70%.
<u>Note</u>: We use the library “org.junit.jupiter:junit-jupiter:5.4.2” which is not previously installed. Thus, to run our tests, go to File -> Project Settings -> Global Libraries. Click on the ‘+’, select ‘From Maven’ and type in the name of this library. Once this library is installed, all the tests should work perfectly!

Our team has refactored code in a meaningful way for our final project stage. Code has been refactored in the following classes:

Within the Manager class we have taken out a lot of code that was congested and hindered the readability and simplicity of our application. In more detail we have taken away a lot of code that could be considered to be bloater. There were irrelevant imports that were taken out, simplification of our variable initializations, and as well as the simplification of our implementation where we were able to take out a major method from our code. A lot of commenting and improvements to the readability of our code for the Manager class was also done.

Notably, the majority of the code in main.java was refactored in the interest of creating a more efficient user interface. Javascript and HTML were used to create a more cohesive user interface that not only provides the user with a more intuitive user experience but also allows us to better adhere to our design principles: the frontend and backend are completely independent of each other.

Our previous Main.java class was also a large bloater (code smell). Having to collect and process user input from the IntelliJ console resulted in very large methods that ended up being hard to work with. Since Main.java also had other responsibilities (e.g. passing collected user input to the backend), it quickly became difficult to make changes to any part of Main.java - even the methods not directly responsible for user input. Now that we’ve created our frontend, we no longer have to worry about a bloated Main.java class - in fact, we no longer needed Main.java at all. Each of the frontend’s pages and responsibilities are now contained in separate Javascript files, and are therefore easy to understand and make changes to if necessary.

Finally, we improved and finalized our project’s comment coverage. Each class is now thoroughly commented with information about who designed and contributed to it, as well as information to help others interpret our code. This was a huge step in terms of improving legibility and accessibility.

Accessibility Report

Equitable Use	- We have made sure that when a user registers, they input their appropriate NTRP player rating, which would ensure that all players regardless of skill level are accounted for. - Useful and marketable to people with diverse abilities.
Flexibility In Use	- One way we could implement this would be to provide a different input method such as using the keyboard to match or pass on potential players. - Another way would be for our app to work on all platforms including on a browser or in an application form for mobile devices.
Simple and Intuitive Use	- Design is easy to understand regardless of the user's experience and knowledge. - Our GUI is very easy to understand with an easy question answer format. Some questions are easier to answer since there is a simple dropdown to select from. - We have made sure that the input for each field is precise and common, for example we cannot type alphabets in the field for phone numbers.
Perception Information	- For individuals who have poor sight, we could implement the ability to change the size of the UI elements to accommodate their needs. - Moreover, much like there are separate modes such as light mode and dark mode, our app can have different tones to accommodate for whatever environment they are using our app in. - Design communicates necessary information effectively to the user, regardless of ambient conditions or the user's sensory abilities.
Tolerance for Error	- Minimizes hazards and the adverse consequence of accidental or unintended actions - For example, we notify users if a username is already taken.
Low Physical Effort	- As mentioned within the flexibility in use section, if a user is on the app for long periods of time, they may resort to using alternative input methods such as the keyboard to do their matching rather than clicking on a mouse. - Can be used efficiently and comfortably and with minimum fatigue.
Size and Space for	- N/A - Size and space for approach and use depend on

Approach and Use	hardware level accommodations which are not applicable to our application.

Target Market

We will market our app called MatchPoint to tennis players. Specifically, tennis players who are looking for rallying partners. Players can register free of charge, and at any NTRP (tennis skill level) to find a partner. Our app aims to bring tennis enthusiasts together across the province, in hopes to remove barriers that might prevent new players from getting involved in the tennis community, and provide skilled learners with a source for competition or friendly rally.

Missed Demographic

Our program will mainly be used by people interested in playing tennis. Therefore, it is less likely for those who have no interest in tennis to use this application. Demographics who do not live in densely populated areas that have tennis courts nearby may be limited in the user base. People who cannot afford to play tennis. Those who are visually impaired or blind will not be able to use the app. Seeing as this demographic is not our target market, and visually impaired folks will be forced to play Tennis instead of Soundball Tennis. Those who are physically disabled would also be placed into the demographic who are not able to fully make use of the application – again Tennis instead of Wheelchair Tennis. To be able to accommodate better for these disabilities, and any disabilities, a toggle button could be integrated into the app to identify requests for para sports. At present, those with disabilities or information they wish to address for their matches can be placed in their profile biography.

Progress Report

Summary of Responsibilities since Phase 1 and significant pull request:

- Alexia:
  - Database Implementation (scratched)
  - API implementation (scratched)
  - Development of Observer and Read - read/write Java to Json
  - Generated Test for Observer and Read within Observer.java
  - Modifications to Player [@Override toString]
  - Link to PullRequest: https://github.com/CSC207-UofT/course-project-slice-n-dice/pull/19
- Aman:
  - Implementation of template design pattern
  - Comments added to classes that I worked on
  - Manager.java refactoring
  - https://github.com/CSC207-UofT/course-project-slice-n-dice/commit/8b051bc94d31e84ead2e1cebb8fa8f793b1c4b16 - Link to the Pull request with refactoring and notes with where contributions are.

- John:
    - Implementation of composite design pattern
    - Research on databases
    - Player.java refactoring
    - Implemented methods within PlayerStats class
    - Links to Pull Requests:
        - https://github.com/CSC207-UofT/course-project-slice-n-dice/pull/20#issue-1075103670
        - https://github.com/CSC207-UofT/course-project-slice-n-dice/pull/17#issue-1073765517
- Mary:
    - GUI research
    - Front-end development (Javascript/HTML); replacement of Main.java
    - .java refactoring
    - Files needed to be directly uploaded to Main, check commits for most recent contributions.
- Poorvi:
    - Completed Testing
    - Created a new class and interface that reads Json to java and sends player statistics to front-end and vice-versa
    - Changes to Player class and implementation of new interfaces
    - .java refactoring
    - https://github.com/CSC207-UofT/course-project-slice-n-dice/pull/7#issue-1068756940
    - https://github.com/CSC207-UofT/course-project-slice-n-dice/pull/21#issue-1075122608
- Rahul:
    - https://github.com/CSC207-UofT/course-project-slice-n-dice/pull/16#issue-1071651160
    - Implementation of template design pattern
        - PrioritySkillManager and PriorityLocationManager
    - Implementation of packaging by feature structure
    - .java refactoring