Poorvi, John, Rahul, Aman, Alexia, Mary
Group-022
CSC207
November 15, 2021

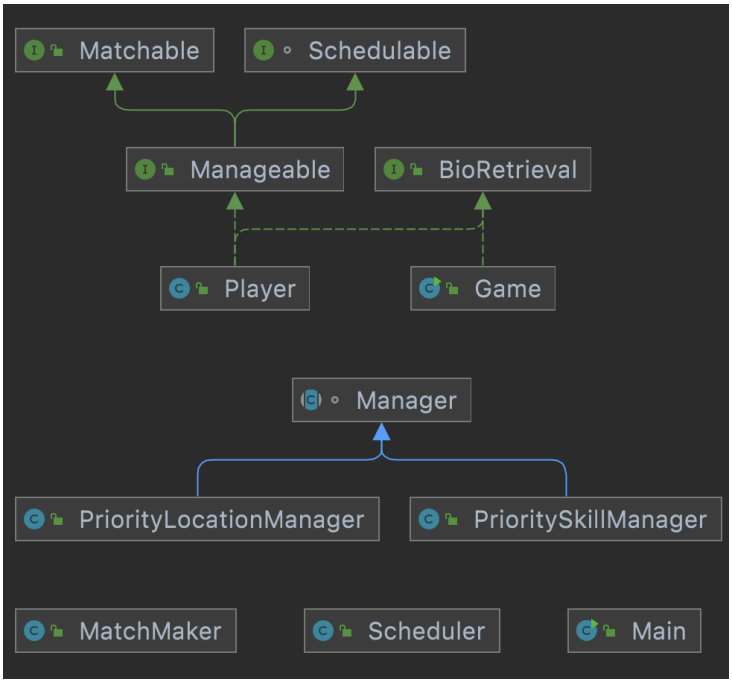
Design Document (Project Phase 1)

Updating Phase 0

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, 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:



Phase 1: Major Design Decisions

In Brief:

- Managing group productivity and accountability via Github
- Validating code against design principles
    - Clean Architecture
    - SOLID principles
- Bringing in new designs for code presentation via design patterns
    - Composite Design
    - Template Design
    - Adapter Design
- Packaging modifications
- Refactoring code

GitHub

Looking towards streamlining our productivity for Phase 1, our group decided to utilize some of the features offered by Github, on our Github page, namely ‘Projects’ and ‘Wiki’. Both ‘Projects’ and ‘Wiki’ are services that allow us to update and track our progress not only across phase 1 but into the completion of the project. ‘Projects’ allows for the development of kanban boards (automated or regular drag and drop) of tasks that can be attributed to members within the group. This ensures that all members and TAs can keep track of to-do, in progress, and finalized processes performed by individual team members. The ‘Projects’ feature’s kanban tool also allows for pull requests, and comments to be addressed and tied to ‘issues’ - to-do’s which require more attention than other cards and have multiple assignees. ‘Issues’ becomes an additional tab through GitHub’s site where developers can quickly access the tasks they are assigned to quickly; avoiding travelling to ‘Projects’ and giving a centralized view of the major assignments they need to work on. The Wiki page allows for all our Google documents to be slotted into the same site where our code can be accessed. In the grand scheme of things, this is a unified bundle of our entire project all localized on Github, rather than dispersed across multiple platforms.

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 don't implement any unnecessary interfaces or depend on random methods. ○ Game implements BioRetrieval ○ Player implements Manageable and BioRetrieval
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 a dependent of all other classes (Scheduler, MatchMaker, and Manager). ● In this way, there is a dependency that goes one direction (for example, Player -> MatchMaker/Scheduler -> Manager). As a result, there is no class that Player depends on, which would be a violation of the dependency inversion principle.

Clean Architecture

- Is your program consistent with Clean Architecture?
    - Describe a scenario walk-through and highlight how it demonstrates Clean Architecture.

Are there any clear violations if we were to randomly look at the imports in a few of your files?

Is the Dependency Rule consistently followed when interacting with details in the outer layer?

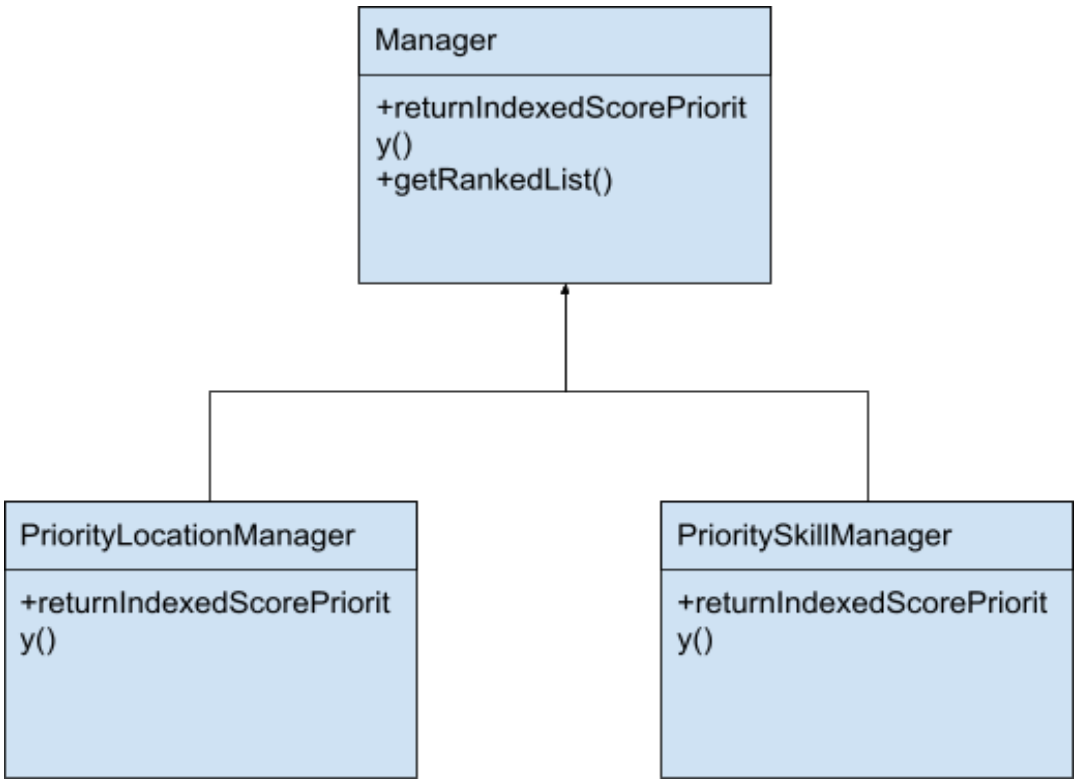
Design Patterns

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



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.

Our Implementation

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 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 this abstract method in such a way that the returned score is weighted more on skill than location.

Packaging Strategy

A smart packaging strategy to use when dealing with multiple entity classes that do not all intertwine, is to separate the packaging by feature, rather than by layer. Packaging by feature is smart because it enables us to package with high cohesion, modularity, and low coupling between packages (which we will not be seeing with this project as we are only working with a singular package looking at the size of Phase 1 and the project thus far). Packaging by feature also enables us, and others reading our program, to navigate our code easier. All the classes that are relative to the same entity class/task are in the same directory. Packing by feature also allows us to understand the applications size much faster, the basic features, and communicate the fundamental aspects of the program faster.

Testing, Refactoring & Code Organization

Most components of our system are not tested. However, our 2 main entity classes are completely tested using JUnit 5. Components that were found most difficult to test include Main. To test our current Main class, we need to provide input from a file. However, as mentioned below, our main class was a major code smell. In phase 2, we will refactor main in a way that it can be tested easily.

Our team has refactored code in a meaningful way for the Phase 1 of our project. Code has been refactored in the following classes: Manager and its variant classes PriorityLocationManager and PrioritySkilledManager, due to the significant modifications the code had to undergo due to change in design to follow better design principles as well as to correct the errors from code written in Phase 0. 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. Now, our main.java class no longer relies on input from lower-level classes - it exclusively collects user input and passes it to the backend.

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. When our frontend and backend merge in Phase 2, Main.java will be very succinct and easy to follow.

Reflection & Next Steps [Progress Report]

Phase 1 Member Responsibilities:

- Alexia:
    - Research on databases
    - Github setup
    - Documentation management
    - Packaging strategy
    - SOLID design principles
- Aman:
    - Implementation of template design pattern
    - .java refactoring
- John:
    - Implementation of composite design pattern
    - Research on databases
- Mary:
    - GUI research
    - Front-end development (Javascript/HTML); replacement of Main.java
    - .java refactoring
- Poorvi:
    - Testing
- Rahul:
    - Implementation of template design pattern
    - .java refactoring

Open questions:

- What would be the most effective way to incorporate a hosting service and database?
- What would be the benefits of having our front-end connect to our API directly?

What worked:

- Delegating responsibilities well
- Time management for assigned tasks

Areas for improvement:

- Variable and method naming. More refactoring to be done before phase 2.
- Communicating prior to deadlines

Next Steps:

- John attended the information session where databases were explored. Alexia and John will continue working towards developing a database potentially on Amazon Web Services (AWS) for Phase 2
- Mary and John will work on the user interface, coding in JS on VSCode with contributions for design coming from all members in iterative waves of development
- Complete testing of all the code
- Implementation of Adapter Design Pattern to connect front-end to back-end
    - Potential for API as back-end
    - JavaScript on VSCode as front-end