3307 Final Submission

Tyler Lin - 251208517 - tlin257

Problem Statement:

Organizing and planning tournaments can be overwhelming. Most of the time tournaments stick to a list of simple structures. This makes planning the matches trivial but tedious. When organizing an event a minimal amount of effort should be put into something so simple. Not to mention when manually setting up matches there is the chance to add human error into the mix. In addition the search and registration of tournaments can also be tedious and time consuming.

My tournament organizer intends to automatically set up matches to cut down on time and confusion. It will also allow managers to find tournaments to join, and free agents to add to their rosters.

System Features:

- ➢ Setting up and organizing a plethora of predesigned tournament structures
  - Swiss, Double Elimination, Single Elimination, Round Robin, Etc.
- ➢ Save records of tournaments
  - Active and inactive
- ➢ User login system for admins, managers, and players
- ➢ Display free agents and their accolades
- ➢ Display open tournaments
  - Registration into tournaments

Design Approach:

- ➢ Encapsulation
  - Will be used to keep information about users separated
  - Will be used to prevent unwanted manipulation of data between tournaments
- ➢ Inheritance
  - A player, manager and admin all inherit from a general user class
  - General user handles logging in, such as username and password.
- ➢ Polymorphism
  - Tournaments will have different matchmaking algorithms based on their type
- ➢ Structure
  - Users have a username and password
  - Players, Admins, and Managers are users
  - All tournaments have list of matches, a name, and a title

- Object Relationships
    - Players are in a team
    - Managers are in charge of a team
    - Managers sign up teams for tournaments
    - Admins make tournaments
    - Tournaments have matches
    - Admins report matches

Requirements:

- Functional
    - Creation of different types of users
    - Creation of different types of tournaments
    - Tournaments will match make games
    - Data must be able to be stored for future uses
- Non-Functional
    - Data encryption for security
    - Open design for future implementations

Purpose:

- Inheritance
    - There are two places I decided to implement inheritance in my project. Users and Tournaments.
    - The reason I decided to implement inheritance is because both Users and Tournaments will have identical attributes and actions regardless of its subtype.
    - For example all Users will need a username, password, and name, and all Tournaments will have an ID, a list of games, etc.
    - By using inheritance instead of multiple classes we can reuse code. This will be better in the long term, if more implementations are added
- Abstraction
    - Abstraction is also used for Tournaments.
    - The reason I chose to implement abstraction is different Tournaments will need to implement the same functions but in different ways.
    - For example, a Swiss tournament would match make games differently than a Single elimination tournament

Pattern Justification:

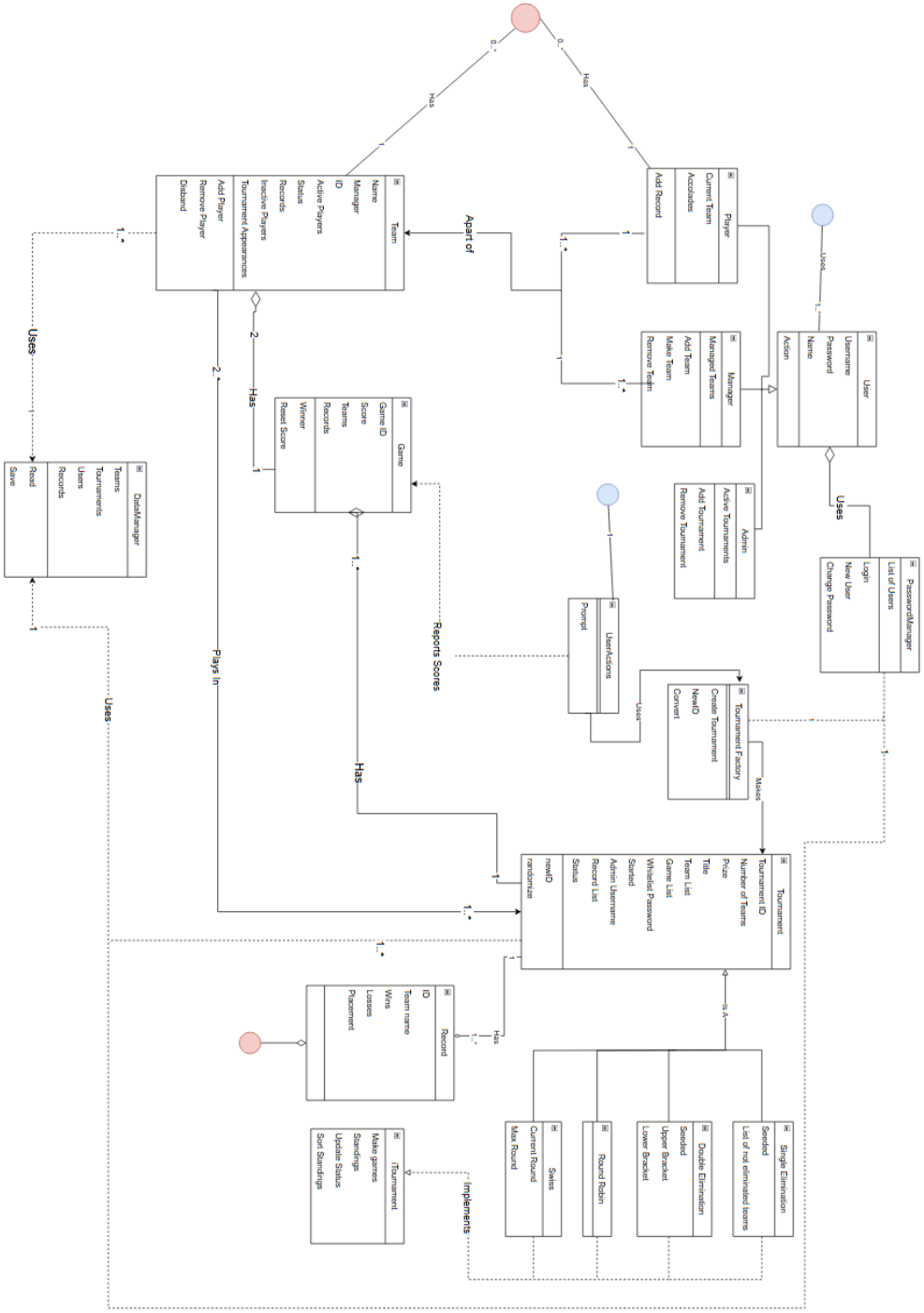
- ➢ Factory Method
  - The factory method will be used to create different types of tournaments
  - Certain tournaments require extra information before they start so a factory in charge of gathering the correct information for the specified tournament is ideal
- ➢ Singleton Pattern
  - The tournament factory will also use the singleton pattern
  - The action manager will also use the singleton pattern
  - Since this is a multi-user program different users will be performing the same tasks
  - However, multiple instances of an unchanged class is a waste of memory.
  - Thus a singleton pattern is ideal
- ➢ Facade Pattern
  - The UserActions class is a facade pattern.
  - It is in charge of prompting Users on what actions to handle.
  - This adds more encapsulation, thus hiding the complexities from users.
  - By implementing this, future versions will have a lot easier of a time expanding
  - For example if we allow Users to be Managers + Players or Admin + Manager.
  - With a single class incharge of serving the menu of options to users is easier
  - NOTE: If I were to redo this project I would still have a facade pattern but offload the functions better than what I have right now
- ➢ NOTE: The following are **NOT** implemented but would be beneficial for future additions
  - Iterator Pattern: There is a lot of code with very similar logic. An iterator would be able to reduce the amount of duplicate code
  - More Factories: Game and Record Factories could significantly reduce the weight that the Tournament class does.

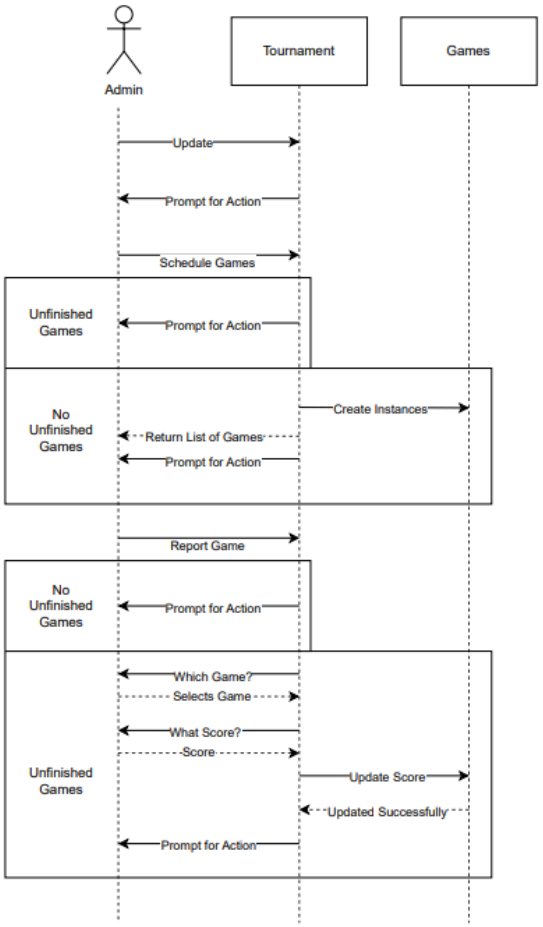
Challenges & Solutions:

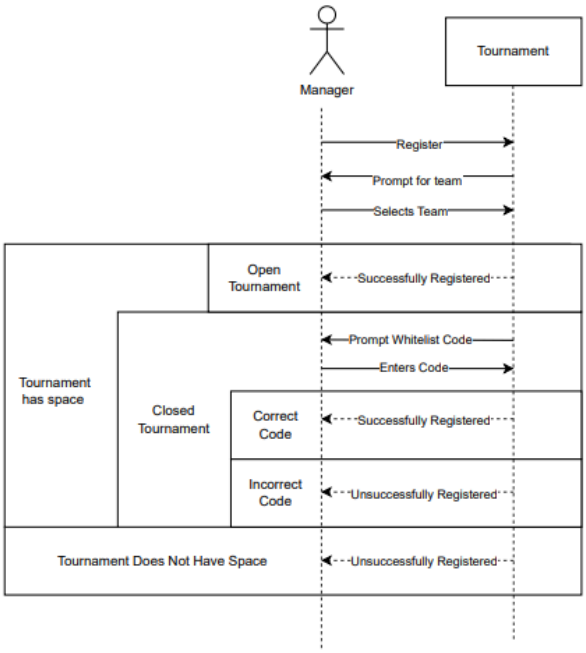
- ➢ Although Tournaments have different methods for matchmaking and standings there are still a lot of common elements between them.
  - I debated between a couple of different methods to maximize reusability of code, but could not figure out a solution better than what I currently have
  - The inherited methods are all the ones in common with each other, and although the logic is very similar between tournaments, the interface is the optimal solution for the slight alterations within the methods
- ➢ Figuring out a way to generate different types of tournaments when they have specific needs was an interesting problem.
  - After researching more about the factory pattern, I found it was a great solution.

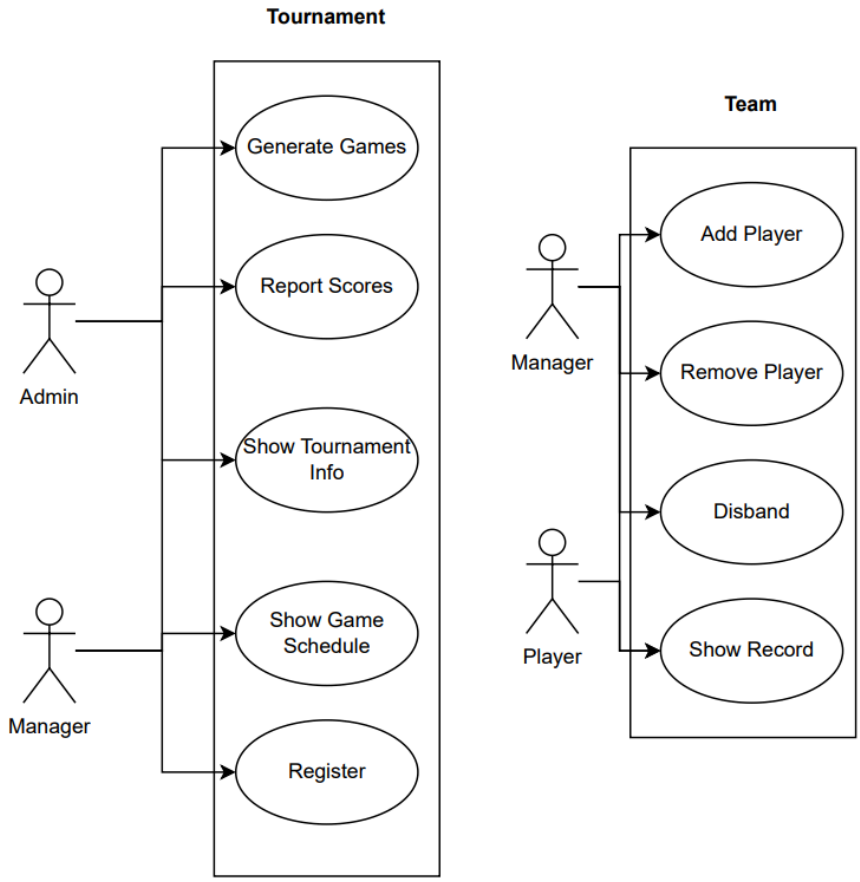
- Algorithm design of specific matchmaking was more difficult than I expected.
    - Visually the matchmaking seems trivial, but translating that to code was harder than expected
    - After research, thinking, and hand drawing my thoughts I figured out all the algorithms needed
- **POST DELIVERABLE 2 CHALLENGES**
    - I decided to create a project that is simply too massive for one person to work on in this timeframe. I am disappointed that what I am handing in is just a fraction of what I wanted, but it is too late to change my project.
    - The way I originally designed the Objects there was tight coupling that created a cycle. Tournaments had Games that had Teams that had Users that had Tournaments. This caused issues when attempting to program JSON storage, as I had to break the link somewhere to read and write the data.

Final UML Diagrams:




Admin
Tournament
Games
Update
Prompt for Action
Schedule Games
Unfinished Games
Prompt for Action
No Unfinished Games
Create Instances
Return List of Games
Prompt for Action
Report Game
No Unfinished Games
Prompt for Action
Unfinished Games
Which Game?
Selects Game
What Score?
Score
Update Score
Updated Successfully
Prompt for Action


Manager
Tournament
Register
Prompt for team
Selects Team
Open Tournament
Successfully Registered
Tournament has space
Prompt Whitelist Code
Enters Code
Closed Tournament
Correct Code
Successfully Registered
Incorrect Code
Unsuccessfully Registered
Tournament Does Not Have Space
Unsuccessfully Registered


Tournament
Generate Games
Report Scores
Show Tournament Info
Show Game Schedule
Register
Admin
Manager
Team
Add Player
Remove Player
Disband
Show Record
Manager
Player