The System for Production Of Recreational Team Scheduling (SPORTS)
Design Document

Authors:
Wendy Meng
Evan Pochtar
Timothy Tu
William Wang

Group: 5

The purpose of this document is to provide you with a guideline for writing the software design document for your project.

Points to remember:

- *Content is important, not the volume.* ***Another team should be able to develop this system from only this document.***
- *Pay attention to details.*
- *Completeness and consistency will be rewarded.*
- *Readability is important.*

This page intentionally left blank.

Document Revision History

Date	Version	Description	Author
11/14/2024	0.0	Initial draft	
11/14/2024	0.1	Finished Section 1	Evan Pochtar
11/14/2024	0.2	Finished Text of Section 2	Evan Pochtar
11/20/2024	0.3	Finished Section 3	Evan Pochtar
11/24/2024	0.4	Added Class Diagram	Wendy Meng
11/24/2024	0.5	Finished Section 4 text and descriptions	Wendy Meng
11/24/2024	0.6	Finished Section 7 and 8	Timothy Tu
11/25/2024	0.7	Added Diagrams for Section 2	Evan Pochtar
11/25/2024	0.8	Finished Section 5 and 6	William Wang
11/25/2024	0.9	Changed and cleaned up Section 2	Evan Pochtar
11/25/2024	0.95	Cleaned up and updated Section 4 text	Wendy Meng
11/25/2024	0.97	Draft Finished, Clean up, grammar checks, diagram checks.	Evan Pochtar
11/25/2024	1.0	Final proofread + checks	Timothy Tu

This page intentionally left blank.

Contents

1 Introduction

This document outlines the design specifications for the SPORTS system, a platform for managing sports leagues, venues, and team registrations. It provides software architectural views, component interactions, and design decisions that will guide the system's implementation.

1.1 Purpose

This design document serves as a blueprint for developers, system architects, and stakeholders involved in the SPORTS system implementation. It elaborates on the technical architecture, component relationships, and design patterns chosen to meet the specified requirements. The document is structured to progress from high-level system overview to detailed design specifications, allowing for clear communication of the system to all stakeholders, as well as providing an overview of the system in general.

1.2 System Overview

The SPORTS system is a platform designed to manage recreational sports leagues and their associated operations. The system serves multiple user types including League managers and administrators, Team managers, Venue operators, Players and Vendors (for concessions). The system operates in a web-based environment, allowing for league registration, venue management, scheduling, and regulation compliance. It's designed to handle multiple sports, leagues, and venues simultaneously while enforcing region specific rules as well as weather dependent scheduling.

1.3 Design Objectives

The SPORTS system is designed to support management of team registrations, game scheduling, and venue bookings across a variety of leagues. The primary design objective is to create a scheduling system that meets the diverse needs of teams, league organizers, and venue managers by implementing both functional and non-functional requirements related to league organization, team management, venue scheduling, and concession regulation.

The system provides a structured approach to handling functional requirements, including:

Concessions: Ensuring venue concession policies are followed based on laws and league constraints.

Storage: Storing league data, team rosters, schedules, and regulation histories.

Authorization: Requiring league-specific payments and authorizations for registrations.

League Eligibility: Restricting league registration based on geographic location.

Game Scheduling: Preventing venue conflicts and adjusting schedules due to weather.

League and Team Management: Allowing administrators to modify league structures, enforce team constraints, and provide league types to accommodate different sports and participant groups, as well as managing limits on team numbers for leagues.

The system addresses several non-functional requirements as well:

Performance: Designed to support up to 500 concurrent users without degradation. Venue scheduling will complete conflict checks within 3 seconds, and other key responses will occur within 2 seconds for managers, officials, and players.

Safety and Security: Sensitive data is secured, and access is restricted based on user roles. All transactions and updates involving sensitive information are authenticated.

Reliability and Scalability: The system aims to have 99.9% uptime and is modularly designed to support the addition of new sports, fields, and league configurations without major overhauls. It can expand to serve multiple cities if needed.

Usability: The interface will be user-friendly, supporting users with varying technical skills, with clear instructions for registration, scheduling, and error handling for conflicts.

Business Rules Compliance: Enforces rules such as role-based permissions for league creation, and automated reminders to update game results within set timeframes.

1.4 References

1) SPORTS System Requirements Specification Document
2) SPORTS Use Cases
3) SPORTS Specifications Document

1.5 Definitions, Acronyms, and Abbreviations

TBD - To be determined, an item which will be determined at a later date after more information is acquired

SPORTS - System for Production Of Recreational Team Scheduling, acronym for the application

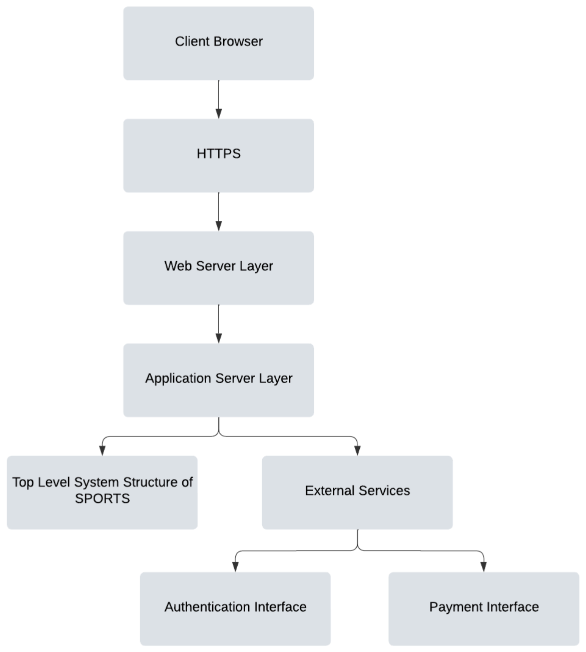
2 Design Overview

2.1 Introduction

The SPORTS System adopts an object-oriented design approach to achieve modularity, reusability, and scalability. The system architecture follows a client-server model that allows users to interact through an online web interface, with the server handling data processing, database interactions, and other backend operations. This model makes sure that the system can handle multiple concurrent users, each accessing league management, scheduling, and user account functions. To develop and document the system, lucidchart is being used for diagramming the architecture, illustrating workflows, and modeling various components of the system.

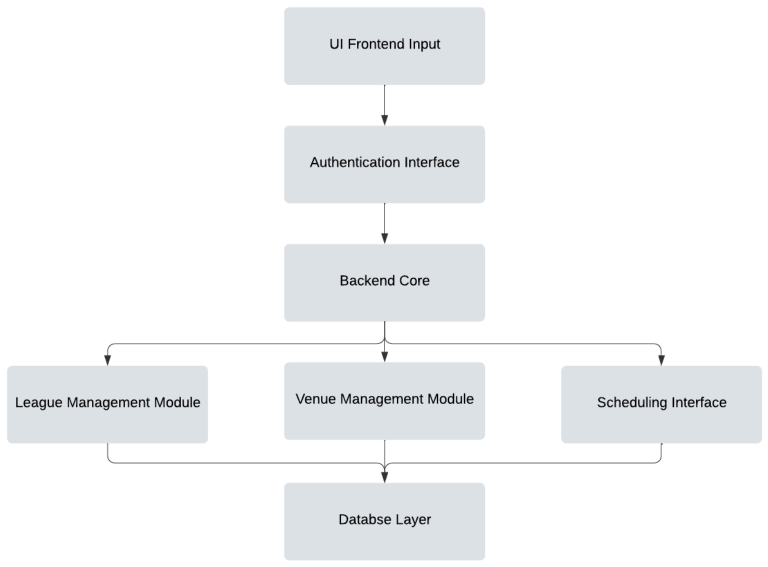
2.2 Environment Overview

The system will operate as a web-based application accessible via browsers. The frontend system from the client will send inputs into the application server layer, which works with the database layer and external services to send an output back to the client's site.



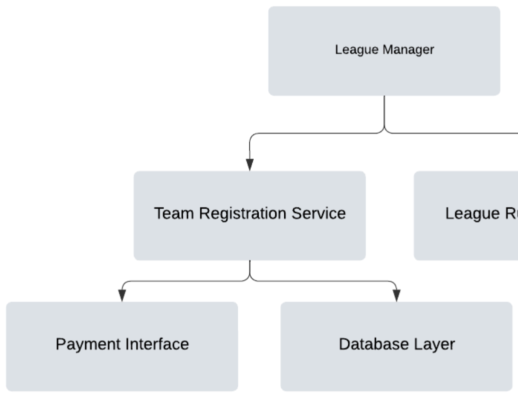
2.3 System Architecture

2.3.1 Top-level system structure of SPORTS



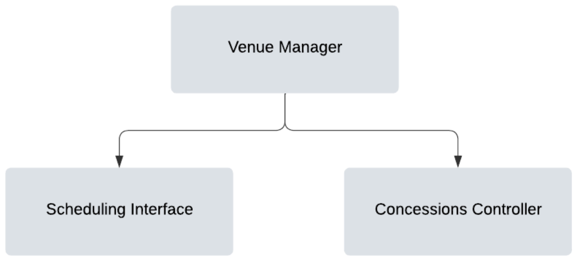
The system consists of three primary modules connected through a main backend core, which allows for the independent scaling of components.

2.3.2 League Management Subsystem



The League Management Subsystem handles team registrations, rule enforcement, and regional eligibility verification. This consists of a League Manager interface, a group of validators, a payment processor, and finally a database layer to hold relevant information.

2.3.2 Venue Management Subsystem



The Venue Management Subsystem coordinates if and when the venues are scheduled, concession management, and the prevention of a venue being double booked.

2.4 Constraints and Assumptions

Constraints

- **Role-Based Access Control**: The system must implement role-based permissions for players, team managers, umpires, Parks & Rec staff, and city officials.

  Design Accommodation: Access control will be implemented using a secure authentication and authorization interface. Each role will have access only to specific functionalities based on their responsibilities.

- **Real-Time Scheduling**: Venue scheduling must occur in real-time, with conflict checks completed within 3 seconds.

  Design Accommodation: Optimized algorithms and database interface queries will be used to ensure fast conflict checking.

- **Payment Processing and Refunds:** Payment errors must be logged immediately, with notifications sent to both users and admins.

  Design Accommodation: Integration with a secure payment gateway will ensure compliant handling of transactions. The system will have thorough error checking in every place an error is possible, and return a frontend message to the user if any errors are thrown.

- **Integration with External Systems:** The system must interact with external scheduling software for seamless booking of fields.

  Design Accommodation: A data gateway that will integrate with external scheduling platforms will be created, ensuring compatibility and data synchronization.

- **Persistent Storage:** Historical league data must be stored in a secure and retrievable manner, even after the league is no longer active.

  Design Accommodation: The system will use a given external database to store and reuse information needed and necessary for all use cases of the site.

Assumptions

- **User Hardware and Internet Access:** Users have access to devices capable of running modern web browsers and reliable internet connections.

  Design Accommodation: The system will prioritize efficient data usage and minimize the amount of calls back to the website for users on slower connections.

- **External System Availability:** External systems like payment gateways and authorization are assumed to be reliable and available.

  Design Accommodation: The system will implement fallback mechanisms and error handling in case of external system downtime.

- **Regulation Updates:** It is assumed that venue managers and city officials will input accurate and up-to-date regulations.

  Design Accommodation: The system will include prompts for stakeholders to review and update regulations regularly.

- **User Training:** Users will receive minimal training and will rely on the system's intuitive interface for guidance.

  Design Accommodation: Usability testing will ensure the interface is intuitive, with tooltips and contextual help provided for complex tasks.

3 Interfaces and Data Stores

3.1 System Interfaces

3.1.1 Scheduler Interface

The Scheduler Interface is used for managing league schedules, including game dates, times, and venues. This is accessible by league admins and team managers via a web-based or mobile GUI. The interface is responsible for the allowing of creation, modification, and deletion of schedules. It also works with the notification subsystem t o provide notifications to teams when a schedule is updated.

Input: Game details (teams, venue, date, time) and rescheduling requests.

Output: Updated schedule, conflict warnings, and notifications sent to teams.

Works with the Game Schedule Data Store to retrieve, update, and store schedule information. Also interfaces with the Notification Subsystem to inform users about schedule changes.

3.1.2 Authentication Interface

The Authentication Interface ensures secure access to the system by verifying user credentials and roles. It provides a user-friendly login and registration GUI for players, admins, and team managers. It includes a single sign on option for easy registration.

Input: Username, password, and 2FA token (if applicable).

Output: Authentication success/failure, user role (e.g., admin, player), or error messages (e.g., invalid password).

Interacts with the User Database to validate credentials and retrieve user roles and permissions.

3.1.3 Payment Interface

The Payment Interface facilitates secure processing of league registration fees, payments for team-related expenses, and other financial transactions. Users (team managers, players, or admins) interact with the payment module through a web-based GUI or mobile app. Payment information, such as credit card details, is captured and securely transmitted The interface also provides feedback, such as success or error messages, after processing payments.

Input: Payment details (e.g., card number, expiration date, CVV, and amount).

Output: Payment status (successful, failed, or pending) and receipts.

Uses external payment gateways (e.g., PayPal, Stripe) via APIs to complete transactions. Logs payment data into the Payment Data Store for auditing and reporting.

3.2 Data Stores

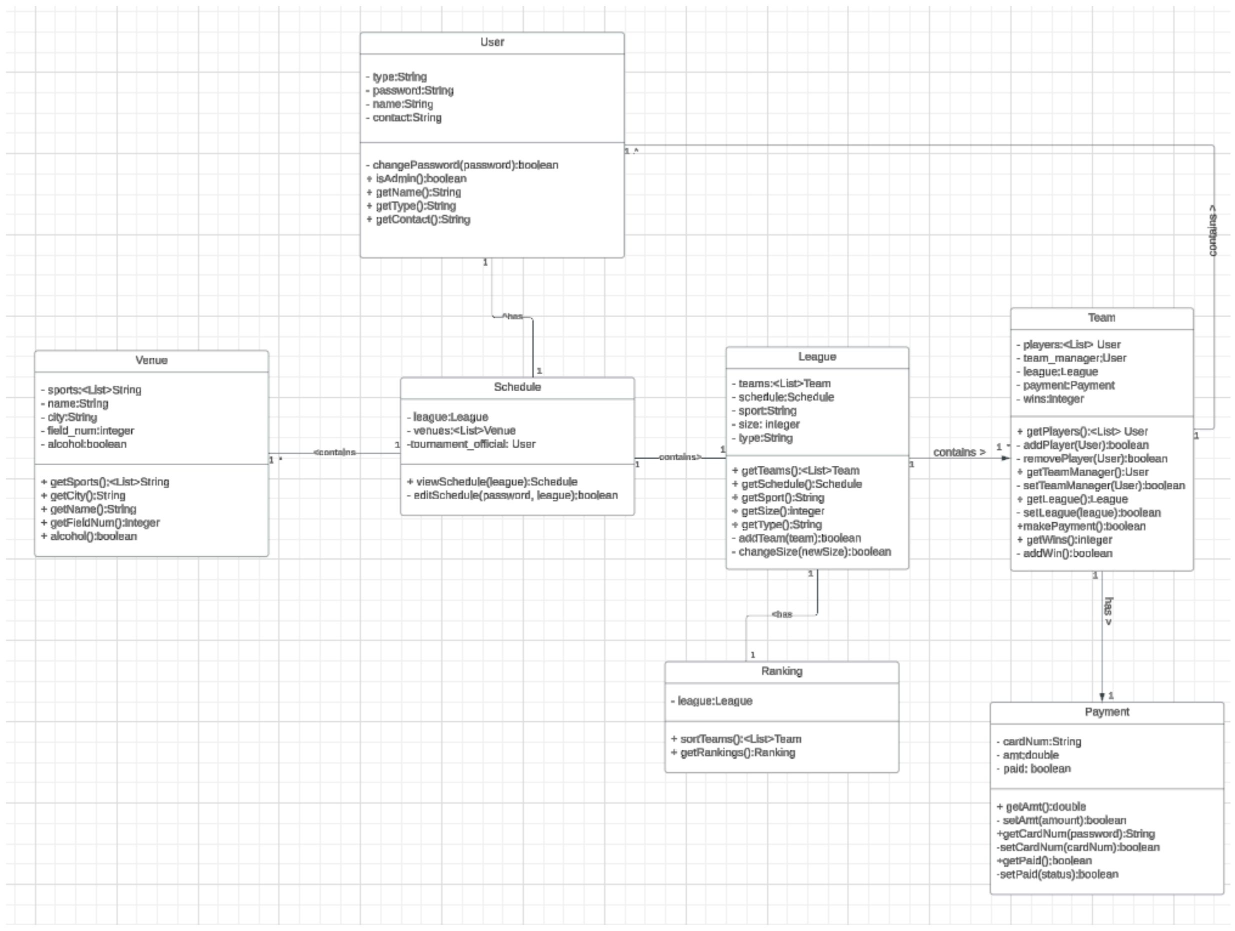
3.2.1 Game Schedule Data Store

This data store holds league schedules, including game dates, times, and venues. Stores game schedules and supports scheduling conflicts resolution, and supplies information to the Notification Subsystem for updates. Key Data Fields:

- Game ID
- Team IDs (home and away)
- Venue
- Date and Time
- Status (scheduled, rescheduled, completed)

4 Structural Design

4.1 Class Diagram

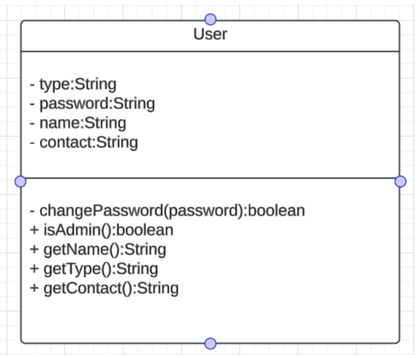


We decided to partition the class diagram into several smaller bits, focusing on specific aspects of the software. For example, a diagram for a user, team, schedule, league, venue, or ranking. This allowed us to hone in closer on specific parts of the system, while still focusing on the overall large functionality through subsystems and top-level system diagrams.

4.2 Class Descriptions

4.3 Classes in the Schedule Subsystem

4.3.1 Class: User



- Purpose: *To organize user accounts and what type of user they are so that they have the correct permissions*
- Constraints: *None*
- Persistent: *Yes (should remain consistent unless someone edits it)*

4.3.1.1 Attribute Descriptions

1. Attribute: *type*
   Type: *string*
   Description: *specifies the type of user(i.e. Admin, city official, etc.)*
   Constraints: *should be a valid user , can't be null*

2. Attribute: password
   Type: *string*
   Description: *password of the user, keeps the user's account safe*
   Constraints: *should have at least one special character and one number and at least 8 characters long, can't be null*

3. Attribute: name
   Type: *string*
   Description: *name of the user*
   Constraints: *can't be null*

3. Attribute: contact
   Type: *string*
   Description: *contact information of the user*
   Constraints: *should be a valid contact, can't be null*

4.3.1.2 Method Descriptions

1. Method: *changePassword(password)*
   Return Type: *boolean*
   Parameters: *password(the user must know their current password to change it)*
   Return value: *success or failure*
   Pre-condition: *user is logged in*
   Post-condition: *none*
   Attributes read/used: *password*
   Methods called: *User.getContact()*

   Processing logic:

   While they are logged in, the User can change their password. They must first input their current password before inputting their new password. The new possword must have at least one special character and number and be at least 8 characters long. If the password is changed successfully, the system will send a message to the contact information stating that the password has been changed and return true. If it was not changed successfully, the system will send a message to the contact information stating that someone tried to change the password and return false.

Test case 1: *Call changePassword(password) with valid current password and valid old password. Expected output is: true*

Test case 1: *Call changePassword(password) with valid current password but invalid new password. Expected output is: false*

Test case 1: *Call changePassword(password) with invalid current password and invalid new password. Expected output is: false*

Test case 1: *Call changePassword(password) with invalid current password but valid new password. Expected output is: false*

2. Method: *isAdmin()*
   Return Type: *boolean*
   Parameters: *none*

Return value: *true or false*
Pre-condition: *none*
Post-condition: *none*
Attributes read/used: *type*
Methods called: *User.getType()*

Processing logic:
getType is called and isAdmin checks if the type is an admin and returns true if it is and false if it isn't.

Test case 1: *Call isAdmin with a user who is an admin Expected output is: true*

Test case 1: *Call isAdmin with a user who is not and admin Expected output is: false*

3. Method: *getName()*
   Return Type: *String*
   Parameters: *none*
   Return value: *name of the user*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *name*
   Methods called: *none*

   Processing logic:
   getName is called and the name of the user is returned

   Test case 1: *Call getName with a user whose name is bill Expected output is: 'bill'*
4. Method: *getType()*
   Return Type: *String*
   Parameters: *none*
   Return value: *Admin, tournament official, umpire, team manager, or player*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *type*
   Methods called: *none*

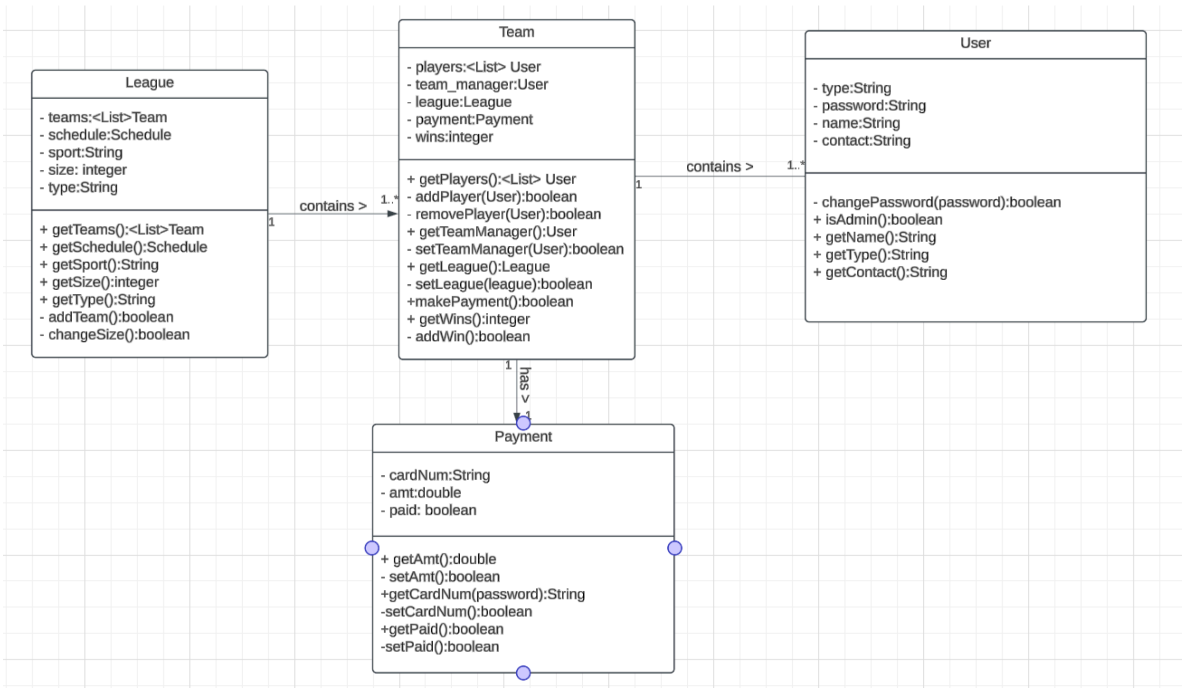
   Processing logic:
   getType is called and the type of the user is returned

   Test case 1: *Call getType with a user who is an admin Expected output is: 'admin'*
2. Method: *getContact()*
   Return Type: *string*
   Parameters: *none*
   Return value: *the contact of the user*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *contact*
   Methods called: *none*

   Processing logic:
   getContact is called and the contact information of the user is returned

   Test case 1: *Call getContact with a user whose contact information is billy@gmail.com Expected output is: 'billy@gmail.com'*

4.3.2 Class: Team



- Purpose: *To manage and contain all the players that are on the same team*
- Constraints: *must have at least one player*
- Persistent: *Yes (should remain consistent unless someone edits it)*

4.3.2.1 Attribute Descriptions

1. Attribute: *players*
   Type: *list of User*
   Description: *contains all the players on the team*
   Constraints: *at least one player*

2. Attribute: team_manager
   Type: *User*
   Description: *the manager of the team*
   Constraints: *can't be null*

3. Attribute: league
   Type: *League*
   Description: *the league that the team is registered for*
   Constraints: *can't be null*

4. Attribute: payment
   Type: *Payment*
   Description: contains the payment method of the team
   Constraints: *none*

5. Attribute: wins
   Type: integer
   Description: *the amount of wins the team has*
   Constraints: *none*

4.3.2.2 Method Descriptions

1. Method: *getPlayers()*
   Return Type: *User list*
   Parameters: *none*

Return value: *a list of all the players*
Pre-condition: *none*
Post-condition: *none*
Attributes read/used: *players*
Methods called: *none*

Processing logic:
getPlayers is called and the list of all the players is returned as a result

Test case 1: *Call getPlayers with players = [bill, alec]. Expected output is: [bill, alec]*

2. Method: *addPlayer(User)*
   Return Type: *boolean*
   Parameters: *User - the player that is to be added to the team*
   Return value: *success or failure*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *players*
   Methods called: *none*

   Processing logic:
   addPlayer is called with a specific player. That player is then added to the list of players

Test case 1: *Call addPlayer with a player, dave, with players = [bill, alec] Expected output is: success and players = [bill,alec,dave]*

3. Method: *removePlayer(User)*
   Return Type: *boolean*
   Parameters: *User- the player that will be removed from the team*
   Return value: *success or failure*
   Pre-condition: *the user a player in the team*
   Post-condition: *the user is not a player in the team*
   Attributes read/used: *players*
   Methods called: *team.getPlayers()*

   Processing logic:
   removePlayer is called with a player in the team. Then getPlayers is called to ensure that the player is in the team. Then the player is removed from the team.

Test case 1: *Call removePlayer with parameter billy when players = [billy, alec]. Expected output is: success and players = [alec]*

Test case 1: *Call removePlayer with parameter billy when players = [alec]. Expected output is: fail and players = [alec]*

4. Method: *getTeamManager()*
   Return Type: *User*
   Parameters: *none*
   Return value: *team manager*
   Pre-condition:there is access to the team
   Post-condition: *none*
   Attributes read/used: *team_manager*
   Methods called: *none*

   Processing logic:
   The value of team_manager is returned for the team.

Test case 1: *Call getTeamManager Expected output is: the value of team_manager*

5. Method: *setTeamManager(User)*
   Return Type: *boolean*

Parameters: *User- the user to be the team manager*
Return value: *success or failure*
Pre-condition: *team exists*
Post-condition: *team_manager is set*
Attributes read/used: team_manager
Methods called: *none*

Processing logic:
The specific team is accessed and the team_manager is set to the User that was passed in

Test case 1: *Call setTeamManager with a valid user. Expected output is: success*

Test case 2: *Call setTeamManager with an invalid user. Expected output is: failure*

6. Method: *getLeague()*
Return Type: *League*
Parameters: *none*
Return value: *league*
Pre-condition: *the team exists and is registered for a league*
Post-condition: *none*
Attributes read/used: *league*
Methods called:

Processing logic:
The attribut league is returned

Test case 1: *Call getLeague Expected output is: the league object*

7. Method: *setLeague(league)*
Return Type: *boolean*
Parameters: *league - the user will specify which league to register for*
Return value: *success or failure*
Pre-condition: team exists
Post-condition: *league attribute is set to the inputted parameter*
Attributes read/used: *league*
Methods called: *none*

Processing logic:
A league is taken in and the team is registered for the league when the league attribute is set to the inputted parameter

Test case 1: *Call setLeague with a valid league . Expected output is: success*

Test case 1: *Call setLeague with a invalid league . Expected output is: failure*

8. Method: *makePayment()*
Return Type: *boolean*
Parameters: none
Return value: *success or failure*
Pre-condition: *payment attribute must be complete*
Post-condition: *payment is made*
Attributes read/used: *payment*
Methods called: *payment.getCardNum, payment.getAmt, payment.setPaid*

Processing logic:
The card number and amount to be paid is retrieved and the system uses the card to pay the specified amount. Then the paid attribute in payment is set to true if the payment is a success.

Test case 1: *Call make Payment with the payment attribute completed Expected output is: success and payment's paid attribute is true*

Test case 2: *Call make Payment without the payment attribute completed Expected output is: failure and payment's paid attribute remains false*

9. Method: *addWin()*
   Return Type: *boolean*
   Parameters: *none*
   Return value: *success or failure*
   Pre-condition: *the team wins the game*
   Post-condition: *attribute wins is incremented*
   Attributes read/used: *wins*
   Methods called: *none*

   Processing logic:
   When the team wins a game, addWin is called and the team's attribute wins is incremented by one

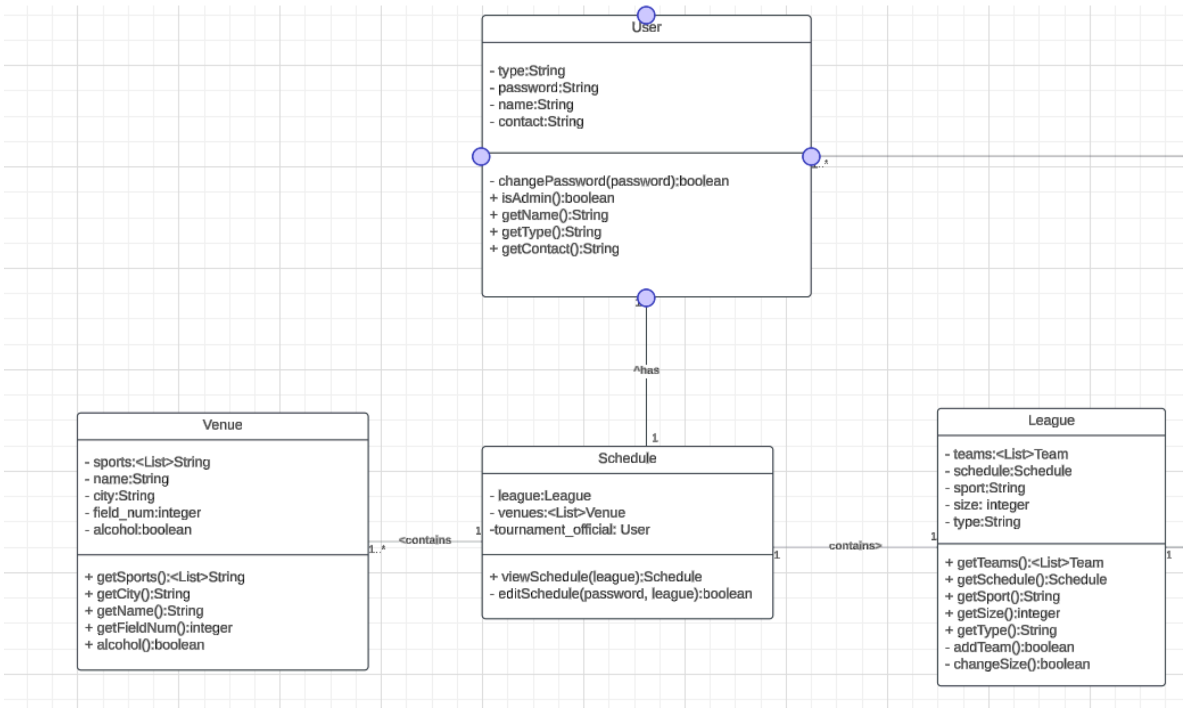
   Test case 1: *Call addWins. Expected output is: wins+1*

10. Method: *getWins()*
    Return Type: *integer*
    Parameters: *none*
    Return value: *wins*
    Pre-condition: *none*
    Post-condition: *none*
    Attributes read/used: *wins*
    Methods called: *none*

    Processing logic:
    The value in wins is returned

    Test case 1: *Call getWins Expected output is: wins*

4.3.3 Class: Schedule



- Purpose: *To organize games for teams in each league at specific locations and days*
- Constraints: *None*
- Persistent: *Yes (should remain consistent unless someone edits it*)

4.3.3.1 Attribute Descriptions

1. Attribute: *league*
   Type: *League*
   Description: *ensures that every team in a specific schedule is in the same league*
   Constraints: *should be a valid league, one offered by the organization*

2. Attribute: tournament official
   Type: *User*
   Description: *a User that officiates the tournament*
   Constraints: *should be a valid user*

3. Attribute: venues
   Type: *Venue list*
   Description: *list of venues that support the league*
   Constraints: *should be compatible with the league and its sport*

4.3.3.2 Method Descriptions

1. Method: *viewSchedule (league)*
   Return Type: *Schedule*
   Parameters: *league - the user will specify which league's schedule they wish to view*
   Return value: *schedule*
   Pre-condition: *the league exists*
   Post-condition: *none*
   Attributes read/used: league
   Methods called:

   Processing logic:

The system searches through the available schedules and returns the one with the specified league

Test case 1: *Call viewSchedule with a specific league. Expected output is: the schedule for that league*

2. Method: *editSchedule(password, league)*
   Return Type: *boolean*
   Parameters: *password-only admins may access this function with their password; league-the user will specify which league's schedule they wish to edit*
   Return value: *success or failure*
   Pre-condition:the user is an admin and the league is valid
   Post-condition: *the schedule is changed*
   Attributes read/used: *league, password*
   Methods called: *User.isAdmin*

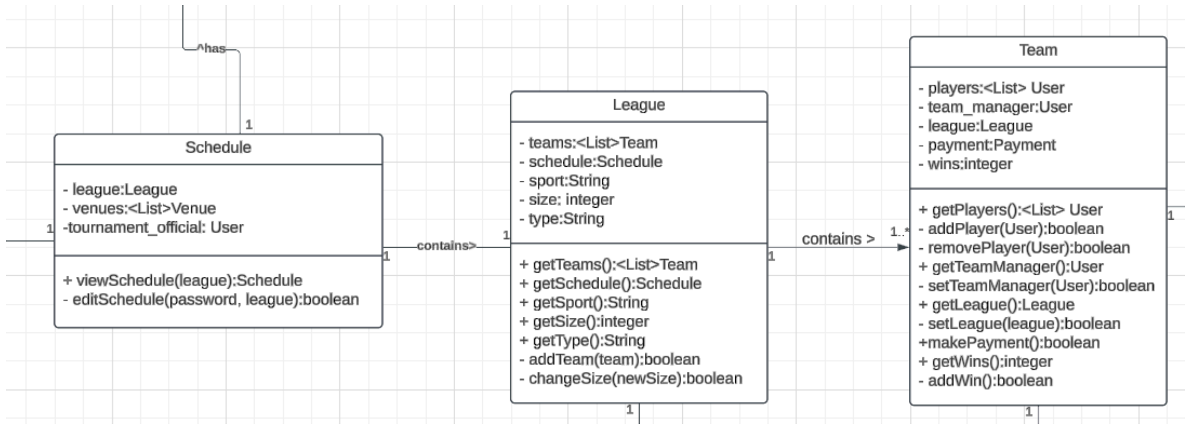
   Processing logic:
   An admin calls the function with a specified league. The system checks if the user is an admin with the passed in password. After confirming, the system find the correct schedule and allows the user to make changes

Test case 1: *Call editSchedule with admin password and valid league and edit an aspect of the current schedule Expected output is: success*

Test case 2: *Call editSchedule with regular user password. Expected output is: failure*

Test case 3: *Call editSchedule with admin password but invalid league. Expected output is: failure*

4.3.4 Class: League



- Purpose: *To contain the teams within the league and the league's information*
- Constraints: *None*
- Persistent: *Yes (should remain consistent unless someone edits it)*

4.3.4.1 Attribute Descriptions

1. Attribute: *teams*
   Type: *Team list*
   Description: *contains all the teams in the league*
   Constraints: *at least one team*

2. Attribute: schedule
   Type: *Schedule*
   Description: *the schedule for the league*
   Constraints: *none*

3. Attribute: sport

Type: *string*
Description: *the sport that the league is for*
Constraints: *should be a sport that the organization offers*

4. Attribute: size
   Type: *integer*
   Description: *the amount of teams that the league can support*
   Constraints: *none*

5. Attribute: type
   Type: *string*
   Description: *the type of league (i.e. women, men, or co-ed)*
   Constraints: *no duplicate types for each sport*

4.3.4.2 Method Descriptions

1. Method: *getTeams()*
   Return Type: *Team list*
   Parameters: *none*
   Return value: *a list of the teams in the league*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *teams*
   Methods called: *none*

   Processing logic:
   The values in the attribute teams is returned

   Test case 1: *Call getTeams. Expected output is: teams*

2. Method: *getSchedule()*
   Return Type: *Schedule*
   Parameters:none
   Return value: *the schedule for the team*
   Pre-condition:none
   Post-condition: *none*
   Attributes read/used: *schedule*
   Methods called: *none*

   Processing logic:
   The object of the attribute schedule is returned

   Test case 1: *Call getSchedule Expected output is: schedule*

3. Method: *getSport()*
   Return Type: *string*
   Parameters: *none*
   Return value: the sport
   Pre-condition:none
   Post-condition: none
   Attributes read/used: *sport*
   Methods called: *none*

   Processing logic:
   The value in the attribute sport is returned.

   Test case 1: *Call getSport with sport = 'soccer' Expected output is: 'soccer'*

4. Method: *getSize()*
   Return Type: *integer*
   Parameters: *none*

Return value: *the size of the league*
Pre-condition: *none*
Post-condition: *none*
Attributes read/used: *size*
Methods called: *none*

Processing logic:
The value of the attribute size will be returned.

Test case 1: *Call getSize with size = 10 Expected output is: 10*

5. Method: *getType()*
   Return Type: *string*
   Parameters: *none*
   Return value: *either women, men, or co-ed*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *type*
   Methods called: *none*

   Processing logic:
   The value of type is returned

   Test case 1: *Call getType Expected output is: value of type (either women, man, or co-ed)*

6. Method: *addTeam(team)*
   Return Type: *boolean*
   Parameters: *team-the team that is to be added to the league*
   Return value: *success or failure*
   Pre-condition: a new team wishes to join a league
   Post-condition: *teams is changed*
   Attributes read/used: *teams*
   Methods called: *none*

   Processing logic:
   Teams will be checked to make sure the new team isn't already in there and then it will add the new team

   Test case 1: *Call addTeam with new team Expected output is: success and team is added to back of teams*

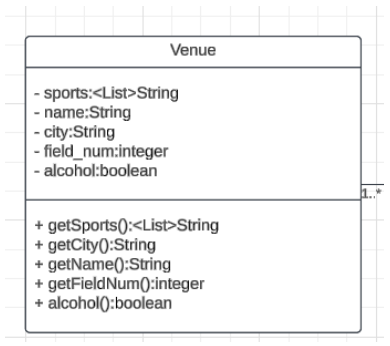
   Test case 2: *Call addTeam with team already in the league Expected output is: failure and teams stays the same*

7. Method: *changeSize(newSize)*
   Return Type: *boolean*
   Parameters: *newSize-the new size of the league*
   Return value: *success or failure*
   Pre-condition: the capacity of the league has changed
   Post-condition: *size is changes*
   Attributes read/used: *size*
   Methods called: *none*

   Processing logic:
   Size is set to the newSize

   Test case 1: *Call changeSize with newSize = 30 Expected output is: success and size= 30*

4.3.5 Class: Venue



• Purpose: *To organize the venues that are available for specific sports as well as the information for those sports*
• Constraints: *None*
• Persistent: *Yes (should remain consistent unless someone edits it)*

4.3.5.1 Attribute Descriptions

1. Attribute: *sports*
   Type:string list
   Description: *a list of sports that can be played at this venue*
   Constraints: *none*

2. Attribute: name
   Type: *string*
   Description:the name of the venue
   Constraints: *none*

3. Attribute: city
   Type: *string*
   Description: *location of the venue*
   Constraints: *none*

4. Attribute: field_num
   Type: *integer*
   Description: *the specific field of the venue*
   Constraints: *none*

5. Attribute: alcohol
   Type: *boolean*
   Description:whether or not alcohol is allowed at the venue
   Constraints: *none*

4.3.5.2 Method Descriptions

1. Method: *getSports()*
   Return Type: *sport list*
   Parameters: *none*
   Return value: *a list of the sports*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *sports*
   Methods called: *none*

   Processing logic:
   The values of the attribute sports is returned

   Test case 1: *Call getSports. Expected output is: sports*

2. Method: *getCity()*
   Return Type: *string*
   Parameters: *none*
   Return value: *city*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *city*
   Methods called: *none*

   Processing logic:
   The value of the attribute city will be returned

   Test case 1: *Call getCity with city = 'Plymouth' Expected output is: 'Plymouth'*
3. Method: *getName()*
   Return Type: *string*
   Parameters: *none*
   Return value: name
   Pre-condition: *none*
   Post-condition: none
   Attributes read/used: *name*
   Methods called: *none*

   Processing logic:
   The value of the attribute name will be returned

   Test case 1: *Call getName with name = 'Lion's Park' Expected output is: 'Lion's Park'*
4. Method: *getFieldNum()*
   Return Type: *integer*
   Parameters: *none*
   Return value: *the field number*
   Pre-condition: *none*
   Post-condition: none
   Attributes read/used: *field_num*
   Methods called: *none*

   Processing logic:
   The value of the attribute field_num is returned.

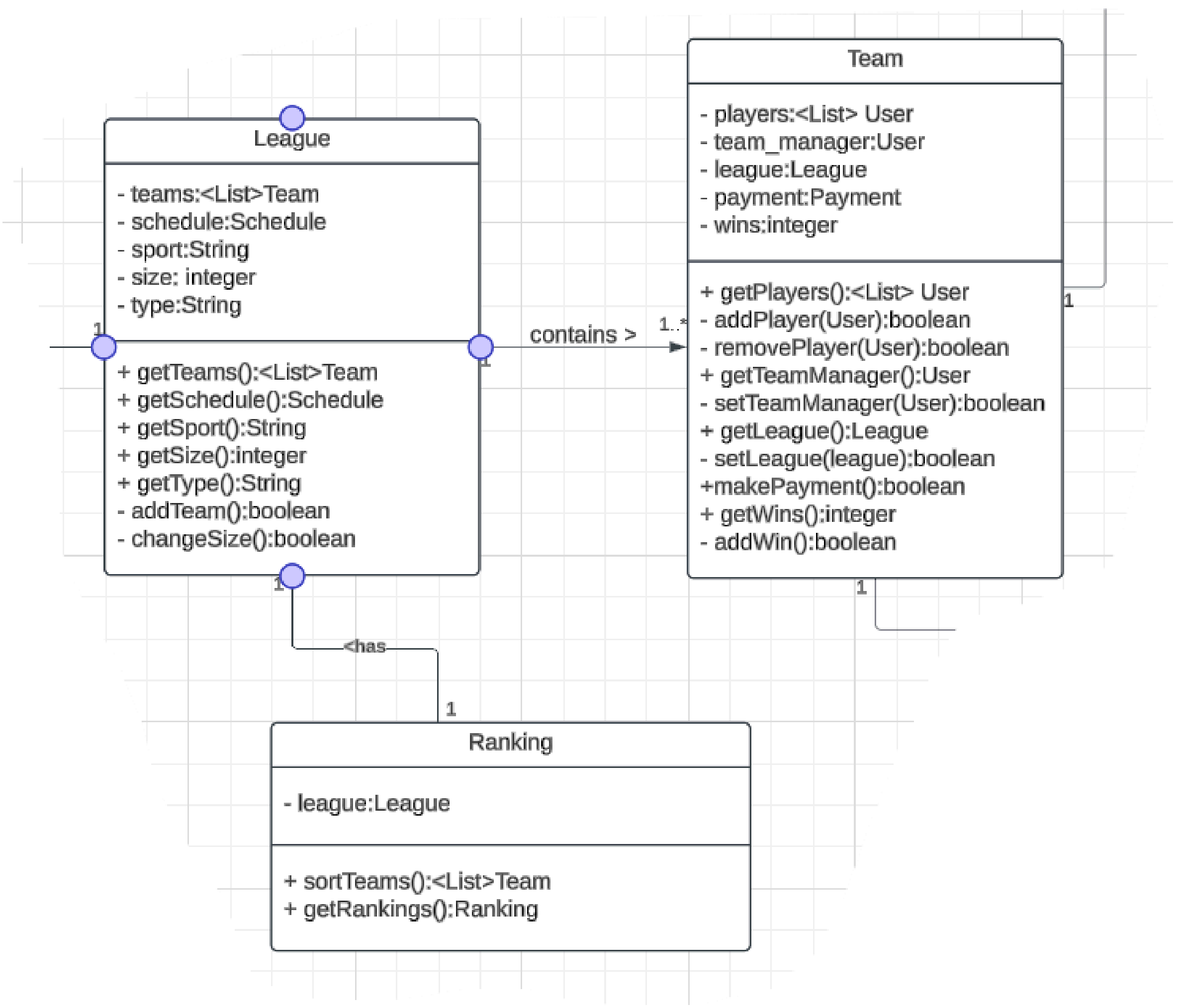
   Test case 1: *Call getFieldNum with field_num = 2 Expected output is: 2*
5. Method: *alcohol()*
   Return Type: *boolean*
   Parameters: *none*
   Return value: *true or false*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *alcohol*
   Methods called: *none*

   Processing logic:
   The value of alcohol is returned

   Test case 1: *Call alcohol when alcohol = true Expected output is: true*

4.4 Class in the Ranking Subsystem

4.4.4 Class: Rankings



- Purpose: *To allow users to see the current rankings of different leagues*
- Constraints: *None*
- Persistent: *Yes (should remain consistent unless someone edits it)*

4.4.1.1 Attribute Descriptions

1. Attribute: *league*
   Type: *League*
   Description: *the specific league that the rankings are on*
   Constraints: *should be a valid league, one offered by the organization*

4.4.1.2 Method Descriptions

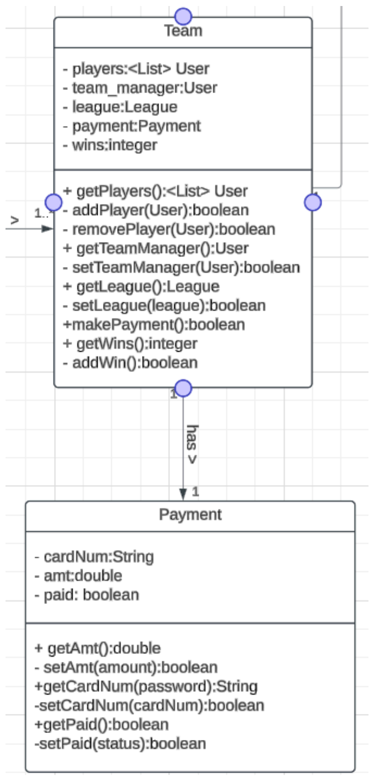
1. Method: *sortTeams()*
   Return Type: *Team list*
   Parameters:none
   Return value: *sorted list of teams*
   Pre-condition: *league is valid*
   Post-condition: *none*
   Attributes read/used: *league, team, User*
   Methods called: *league.getTeams, team.getWins*

   Processing logic:
   The teams will be ordered based on their wins. Putting the team with the most wins in the league first in the list and the one with the least amount of wins last. If there are multiple teams with the same wins, the team that registered first will be placed first.

Test case 1: *Call sortTeams. Expected output is: a sorted list of teams from highest win amount to least*

4.5 Class in the Team Subsystem

4.5.1 Class: Payment



- Purpose: *To keep track of payment information and if payments have been made or not*
- Constraints: *None*
- Persistent: *Yes (should remain consistent unless someone edits it)*

4.5.1.1 Attribute Descriptions

1. Attribute: *cardNum*
   Type: *string*
   Description: holds the card number for the payment
   Constraints: *must be a valid card*

2. Attribute: amt
   Type: *double*
   Description: *amount that is owed*
   Constraints: *none*

3. Attribute: paid
   Type: *boolean*
   Description: *status of the payment*
   Constraints: *none*

4.5.1.2 Method Descriptions

1. Method: *getAmt()*
   Return Type: *double*
   Parameters: *none*

Return value: *amount owed*
Pre-condition: *none*
Post-condition: *none*
Attributes read/used: *amt*
Methods called: *none*

Processing logic:
The value in the attribute amt is returned

Test case 1: *Call viewSchedule. Expected output is:*

2. Method: *setAmt(amount)*
   Return Type: *boolean*
   Parameters: *amount-how much is being paid*
   Return value: *success or failure*
   Pre-condition: *a payment is wished to be made*
   Post-condition: *amount is set to a new amount*
   Attributes read/used: *amt*
   Methods called: *none*

   Processing logic:
   The amount taken in is subtracted from the amt that is owed and the amt that is owed is updated. If the amount taken in is more than the amt owed, set amt to 0.

Test case 1: *Call setAmt with amount = 30 and amt = 30 Expected output is: success and amt = 0*

Test case 2: *Call setAmt with amount = 40 and amt = 30 Expected output is: success and amt = 0*

Test case 2: *Call setAmt with amount = 20 and amt = 30 Expected output is: success and amt = 10*

3. Method: *getCardNum(password)*
   Return Type: *string*
   Parameters: *password-the password of the owner of the payment method*
   Return value: card number
   Pre-condition: *a valid password is passed in*
   Post-condition: none
   Attributes read/used: *cardNum*
   Methods called: *none*

   Processing logic:
   The password is checked to see if it has the right access to the card number, if it does then the card number will returned, if not 0000-0000-0000 will be returned

Test case 1: *Call getCardnum with valid password Expected output is: cardNum*

Test case 2: *Call editSchedule with invalid password. Expected output is: '0000-0000-0000'*

4. Method: *setCardNum(cardNum)*
   Return Type: *boolean*
   Parameters: *cardNum-a valid card number*
   Return value: *success or failure*
   Pre-condition: *none*
   Post-condition: *cardNum is set to a new cardNum*
   Attributes read/used: *cardNum*
   Methods called: *none*

   Processing logic:
The passed in parameter is checked to see if it is a valid card number and if it is then cardNum will be set to it, if not false will be returned.

Test case 1: *Call setCardNum with valid card number Expected output is: success and cardNum is set to new card number*

Test case 2: *Call setCardNum with invalid card number Expected output is: failure*

5. Method: *getPaid()*
   Return Type: *boolean*
   Parameters: *none*
   Return value: *true or false*
   Pre-condition: *none*
   Post-condition: *none*
   Attributes read/used: *paid*
   Methods called: *none*

   Processing logic:
   The value of paid is returned

   Test case 1: *Call getPaid Expected output is: value of paid (either true or false)*
6. Method: *setPaid(status)*
   Return Type: *boolean*
   Parameters: *status-whether or not it has been paid*
   Return value: *success or failure*
   Pre-condition:none
   Post-condition: *paid is changed*
   Attributes read/used: *paid*
   Methods called: *none*

   Processing logic:
   Status is either true or false, if the payment has been made or not. Paid will be set to the value of status.

   Test case 1: *Call setPaid with status = true Expected output is: success and paid = true*

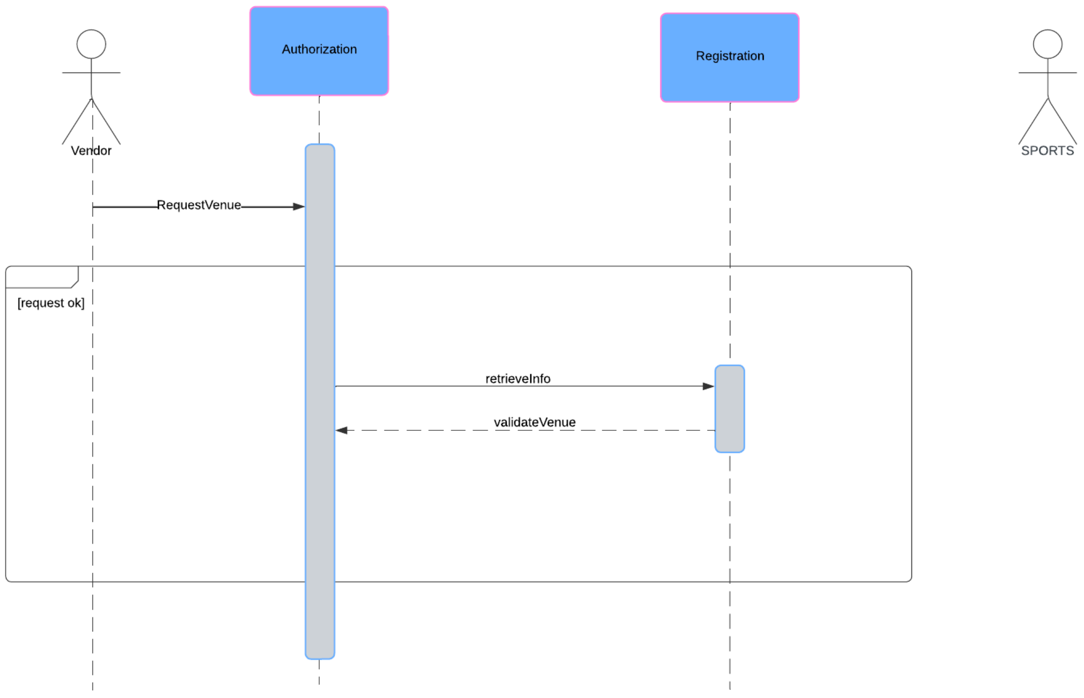
   Test case 2: *Call setPaid with status = false Expected output is: success and paid = false*

5 Dynamic Model

5.1 Scenarios

Scenario: Concession Regulation

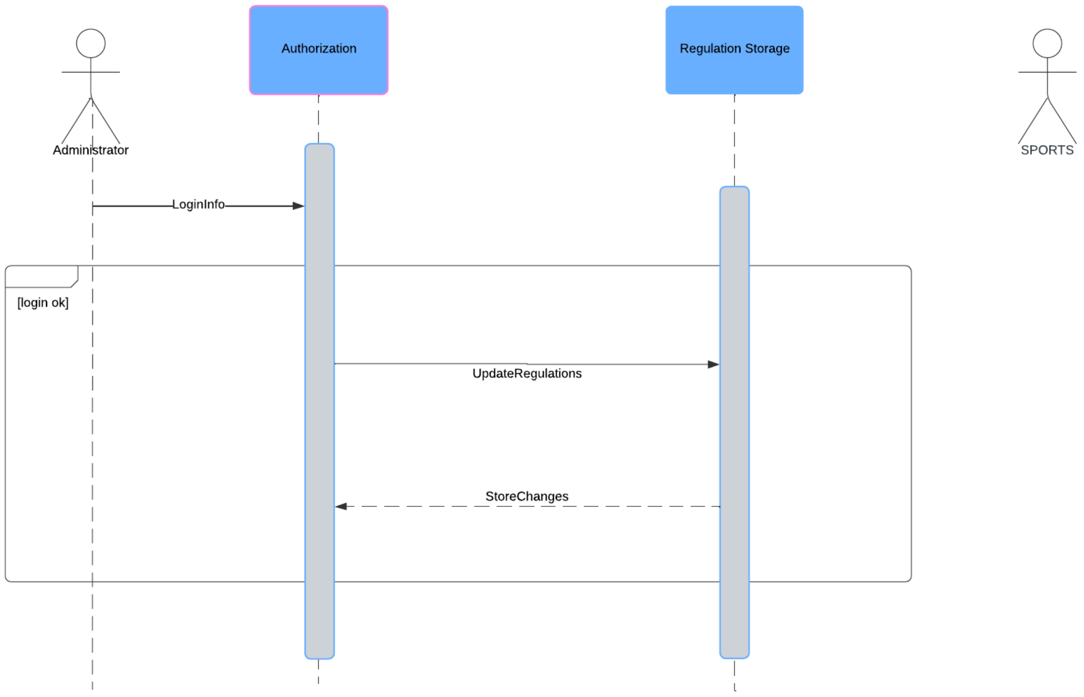
- The vendor requests approval of the sale of an item at a venue from the system. The system will notify the vendor if approval is granted or rejected.



-

Scenario: Inputting Regulations

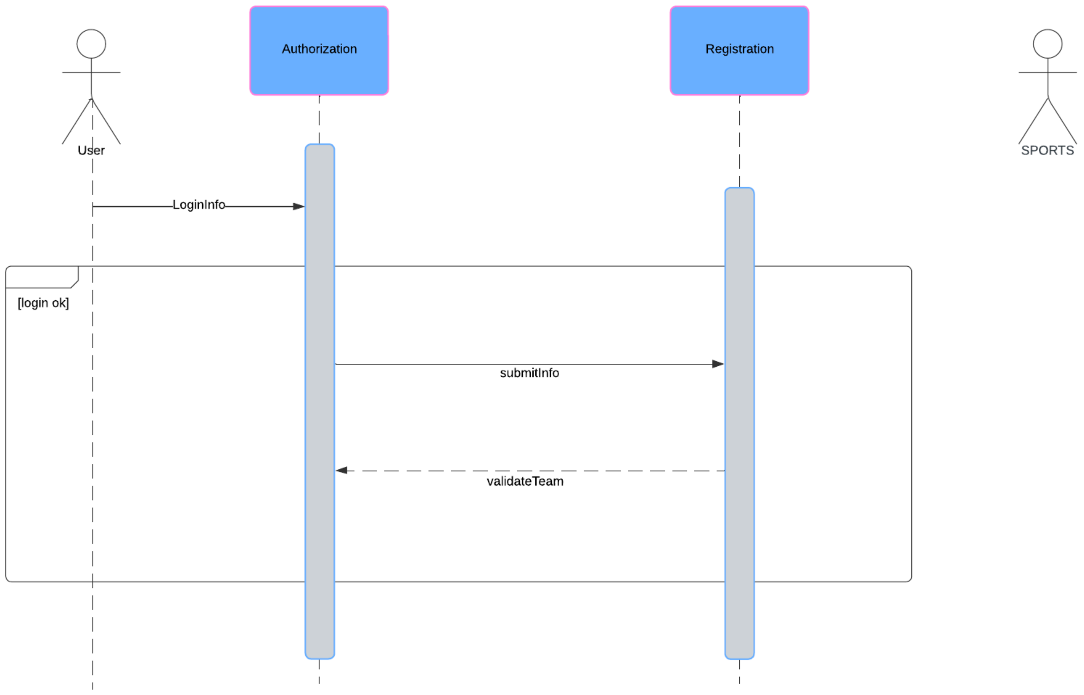
- An administrator adds or changes league regulations. The system will store regulations in persistent storage and monitor changes made.



-

Scenario: Team Information

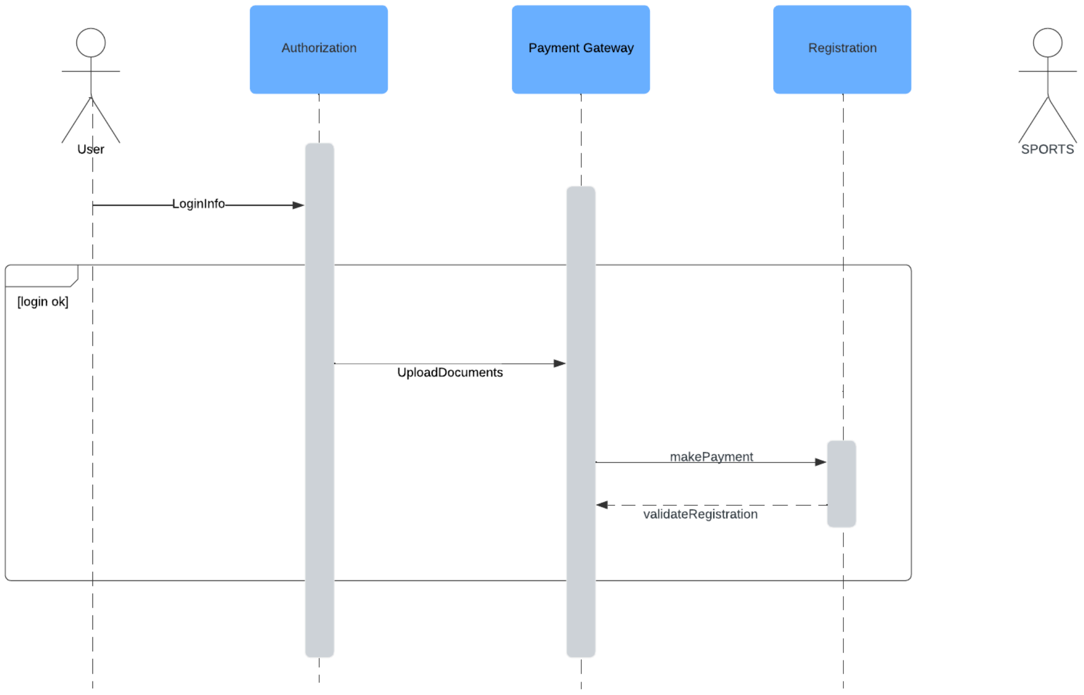
- User signs in and selects a league to join. The user then inputs their team's info and submits it to join the league.



-

Scenario: Making Payments

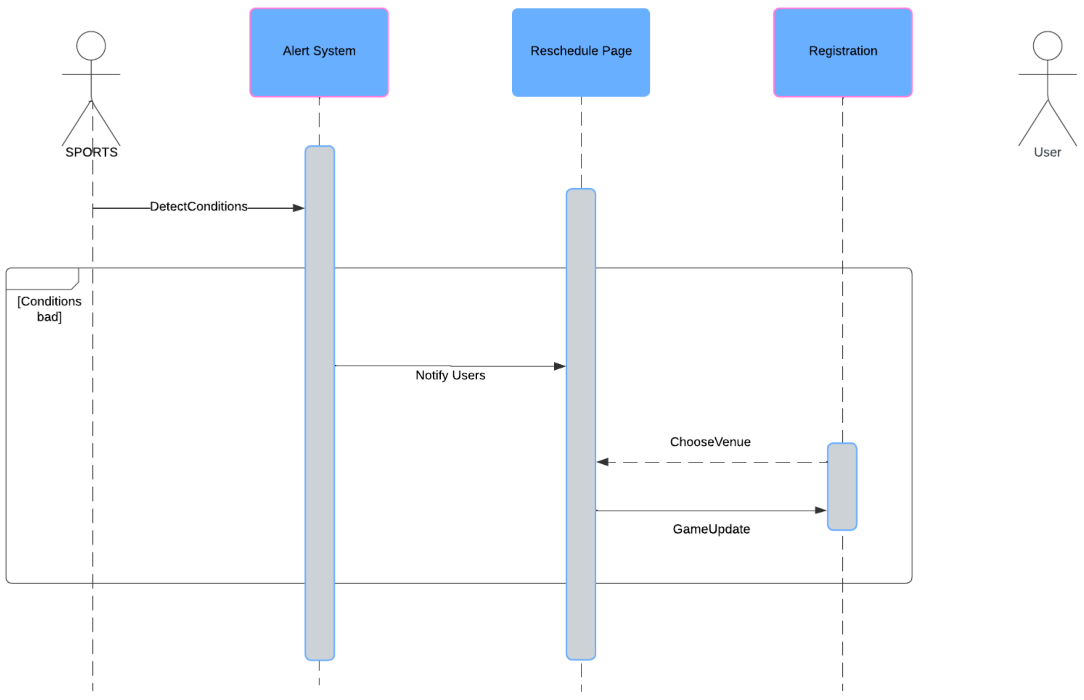
- The user registers for a league. Afterwards, the system asks for authorization which is when the user uploads the required documents. Then the user will make the payment.



-

Scenario: Updating Games

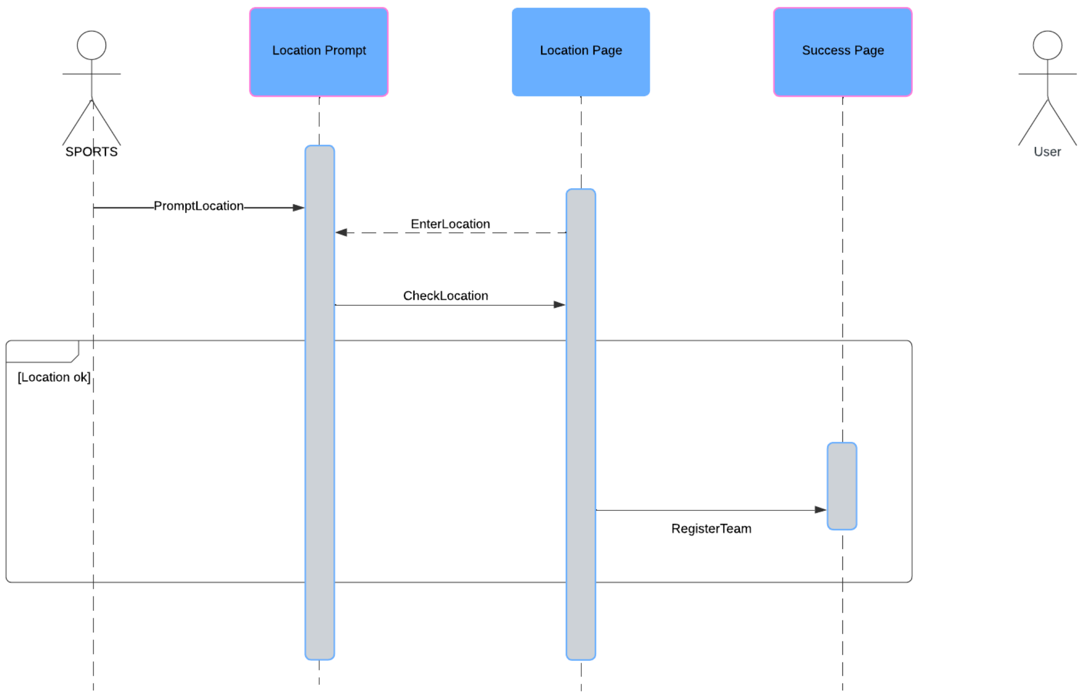
- The system monitors weather conditions and then notifies users if there is a conflict. The user chooses a new date and the game is rescheduled



-

Scenario: Game Eligibility

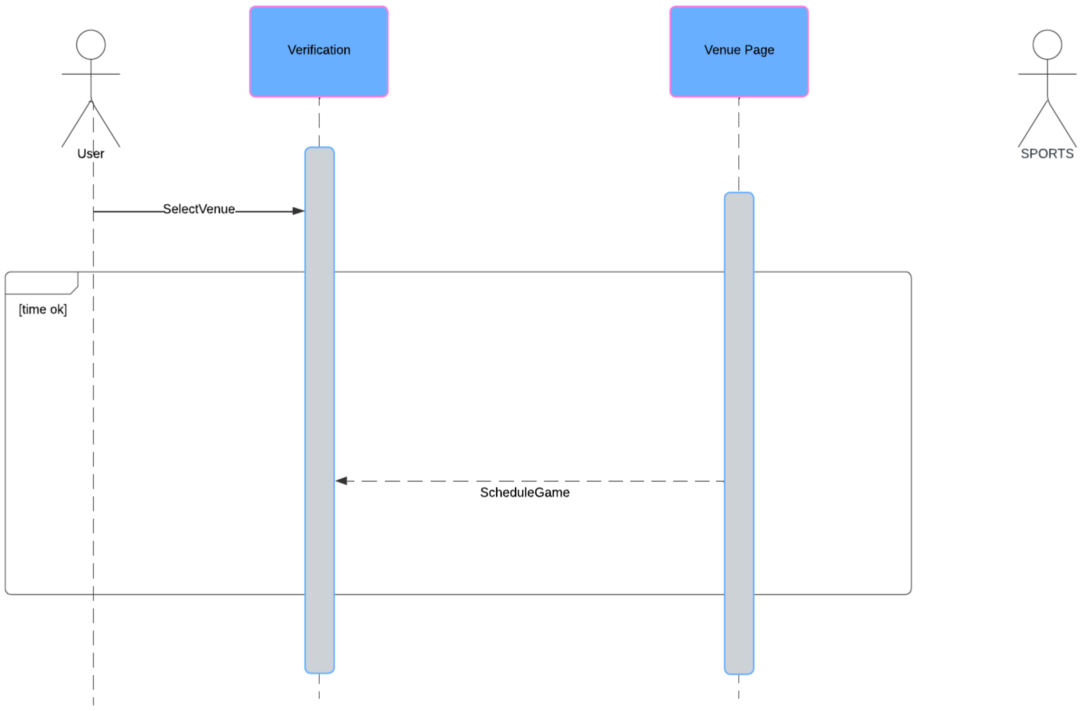
- User registers the team for a league which causes the system to prompt the user for location. The system will check that the region is available and registers if successful



-

Scenario: Venue Conflict

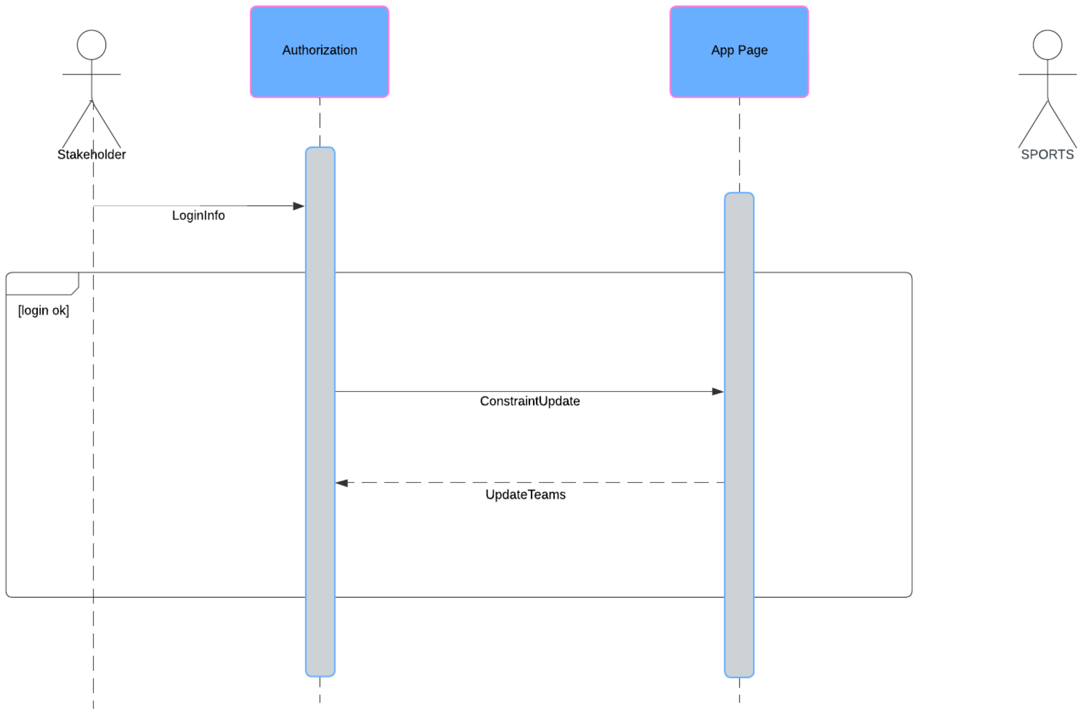
- The user selects a league. The user then selects a time and venue from a list which the system will check for conflicts. If none are found, the game is scheduled.



-

Scenario: Manage teams

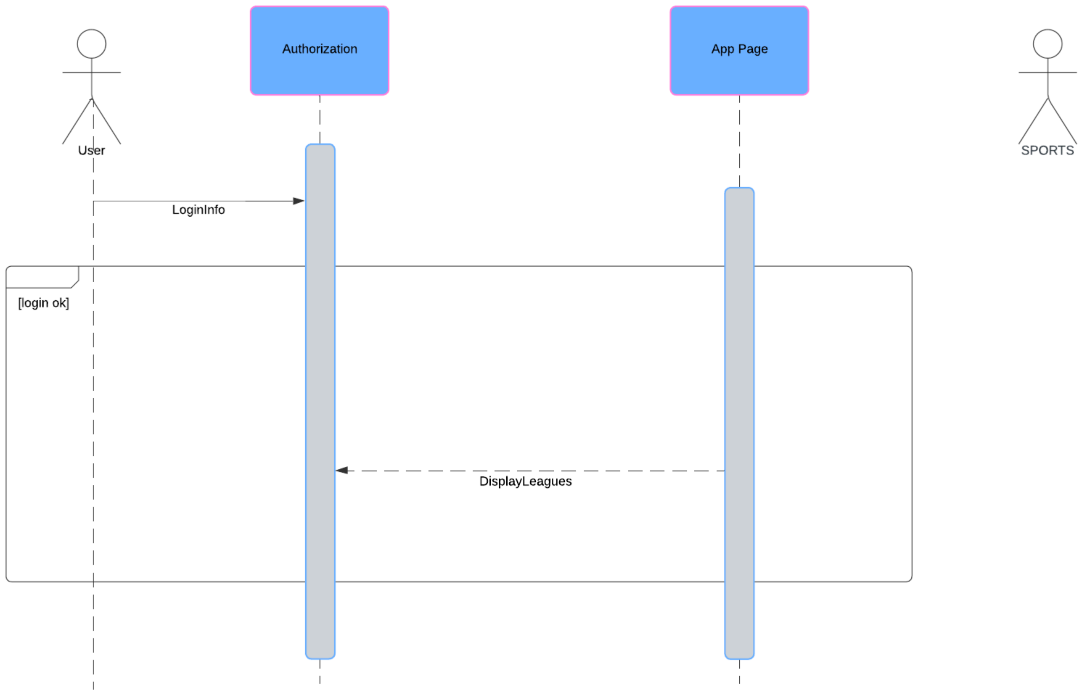
- The stakeholder goes into the app. They can create new constraints or modify old ones applying to teams



-

Scenario: League Variety

- The user goes into the app, and the app should display a wide variety of sports and leagues that are available.



•

6 Non-functional requirements

Performance Requirements

The system will be utilizing a database of users in order to ensure persistent storage of data. Performance cannot be entirely determined by the system so updates within a minute cannot be guaranteed. As the application becomes more mainstream, more updates and changes can be made in the future to accommodate better real time updates.

Safety Requirements

Any sensitive data will be encoded using ciphers and keys that are only accessible to administrators and users who need it. The administrator can add update banners that notify users of game schedule changes or conflicts. In the future, the system will link to a local weather service to tell weather without input. It will also notify users without manual interference necessary.

Security Requirements

The system will automatically assign new users as players. This role can be changed via an access code to gain more permissions such as to team captain. Data retention policies will have due dates to notify administrators to ensure that the system is complying with current policies. The system shall be able to automatically check for secure data retention in the future and notify administrators if issue(s) are found.

Software Quality Attributes

The sports scheduling system will provide a user-friendly platform that connects leagues across multiple cities. It will have an intuitive interface that allows easy scheduling and management of sports events, with the ability to quickly add new sports or expand to new locations. The system will be designed to handle increasing numbers of users, sports, and leagues.

Business Rules

The system will implement role-based access control set by the administrators, restricting league creation to Parks & Rec staff while enabling team managers to register and pay fees. Umpires will be required to submit within 24 hours with notifications for missed deadlines. Concession listings will require city official pre-approval and be scheduled separately from game times. A built-in payment function will allow teams to request payment plans or refunds, with automated processing to minimize manual intervention and streamline the administrative workflow. The system will be able to handle larger venues as well as a more diverse variety of concessions. A matching algorithm will automatically place teams in leagues based on player availability and field resources, directing free agents to teams needing members.

7 Requirements Traceability Matrix

Design Element	Requirement Identifier’s
Concession Regulation Interface	REQ-4.1.1, REQ-4.1.2, REQ-4.1.3
Persistent Storage	REQ-4.2.1, REQ-4.2.2, REQ-4.2.3
League Registration Interface	REQ-4.3.1, REQ-4.3.2, REQ-4.3.3
Payment Processing Interface	REQ-4.4.1, REQ-4.4.2, REQ-4.4.3
Location Verification Module	REQ-4.5.1, REQ-4.5.2, REQ-4.5.3
Game Scheduling Engine	REQ-4.6.1, REQ-4.6.2, REQ-4.6.3
League Management Interface	REQ-4.7.1, REQ-4.7.2, REQ-4.7.3
League Variations Feature	REQ-4.8.1, REQ-4.8.2, REQ-4.8.3
Weather Monitoring and Rescheduling	REQ-4.9.1, REQ-4.9.2, REQ-4.9.3

8 Appendices

Appendix A: Glossary of Terms

This section is for terms used in the Design.

- **League Manager**: An individual responsible for overseeing league operations, including scheduling and compliance with regulations.

- **Role-Based Access Control (RBAC)**: A system that restricts access based on user roles.
- **Venue Scheduling**: The process of assigning and managing game times and locations for sports leagues.
- **Concession Management**: The regulation and administration of food, beverages, and other amenities at sports venues.
- **Payment Gateway**: A service provider that processes secure financial transactions.
- **Weather API**: An external service used to monitor weather conditions and adapt game schedules accordingly.

Appendix B: Class Diagram Key

This section provides a legend for interpreting the class diagrams included in the document.

- **Solid Lines**: Represent associations between classes.
- **Dashed Lines**: Represent dependencies between classes.
- **Filled Diamonds**: Represent composition relationships.
- **Hollow Diamonds**: Represent aggregation relationships.
- **Arrowheads**: Indicate directionality in inheritance or implementation.

Appendix C: Data Field Constraints

A summary of data field constraints applied throughout the system:

- **Username**: Must be unique, non-empty, and 3–30 characters long.
- **Password**: Must be at least 8 characters long, including one special character and one numeric digit.
- **Email**: Must be a valid format (e.g., user@example.com).
- **Date and Time**: Must conform to ISO 8601 standards.

Appendix D: Use Case Summaries

Detailed summaries of the primary use cases covered by the SPORTS system:

1. **Team Registration**:
    - Actors: Team managers, players.
    - Goal: Register a team, pay league fees, and verify eligibility.
    - Key Steps: User authentication, league selection, payment processing.
2. **Game Scheduling**:
    - Actors: League admins, team managers.
    - Goal: Schedule games without conflicts, taking into account weather and venue availability.
    - Key Steps: Select league, assign venue, notify stakeholders.
3. **Weather Monitoring and Rescheduling**:
    - Actors: League admins.
    - Goal: Automatically reschedule games based on adverse weather conditions.
    - Key Steps: Integrate weather API, notify teams of changes.

Appendix E: Error Handling

Explanation of error-handling mechanisms:

- **Payment Errors**:
    - Log errors immediately.
    - Notify the user and admin.
    - Provide the option to retry or request assistance.
- **Schedule Conflicts**:
    - Alert the admin of the conflict.

    - Suggest alternative times or venues based on availability.
- **Authentication Failures**:
    - Limit login attempts to three before locking the account for 15 minutes.
    - Notify the user of the failure and provide password recovery options.

Appendix F: Future Work

Recommendations for extending the SPORTS system:

1. **Mobile Application**: Develop a dedicated mobile app to enhance user accessibility.
2. **Machine Learning Integration**: Use AI to predict scheduling conflicts and optimize league operations.
3. **Expanded User Roles**: Include roles such as spectators and vendors to broaden system functionality.